Partial Answers for Unavailable Data Sources*

Philippe Bonnet[1] and Anthony Tomasic[2]

[1] GIE Dyade, INRIA Rhône Alpes
655 Avenue de l'Europe, 38330 Montbonnot, France
Philippe.Bonnet@dyade.fr
[2] INRIA, Domaine de Voluceau
Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
Anthony.Tomasic@inria.fr

Abstract. Many heterogeneous database system products and prototypes exist today; they will soon be deployed in a wide variety of environments. Most existing systems suffer from an *Achilles' heel*: they ungracefully fail in presence of unavailable data sources. If some data sources are unavailable when accessed, these systems either silently ignore them or generate an error. This behavior is improper in environments where there is a non-negligible probability that data sources cannot be accessed (e.g., Internet). In case some data sources cannot be accessed when processing a query, the complete answer to this query cannot be computed; some work can however be done with the data sources that are available. In this paper, we propose a novel approach where, in presence of unavailable data sources, the answer to a query is a *partial answer*. A partial answer is a representation of the work that has been done in case the complete answer to a query cannot be computed, and of the work that remains to be done in order to obtain this complete answer. The use of a partial answer is twofold. First, it contains an *incremental query* that allows to obtain the complete answer without redoing the work that has already been done. Second, the application program can extract information from a partial answer through the use of a secondary query, which we call a *parachute query*. In this paper, we present a framework for partial answers and we propose three algorithms for the evaluation of queries in presence of unavailable sources, the construction of incremental queries and the evaluation of parachute queries.

1 Introduction

Heterogeneous databases provide declarative access to a wide variety of heterogeneous data sources. Research into improving these systems has produced many new results which are being incorporated into prototypes and commercial products. A limiting factor for such systems however, is the difficulty of providing responsive data access to users, due to the highly varying response-time and availability characteristics of remote

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

T. Andreasen, H. Christiansen, and H.L. Larsen (Eds.): FQAS'98, LNAI 1495, pp. 43–54, 1998.

data sources, particularly in a wide-area environment [1]. Data access over remote data sources involves intermediate sites and communication links that are vulnerable to congestion or failures. Such problems can introduce significant and unpredictable delays in the access of information from remote sources.

In cases data from a remote source is delayed for a period of time which is longer than the user is willing to wait, the source can be considered unavailable. Most heterogeneous database systems fail ungracefully in presence of unavailable data sources. They either assume that all data sources are available, report error conditions, or silently ignore unavailable sources.

Even when one or more needed sites are unavailable, some useful work can be done with the data from the sites that are available. We call a *partial answer* the representation of this work, and of the work that remains to be done in order to obtain the complete answer. In this paper, we describe an approach [2], where in presence of unavailable data sources, a *partial answer* is returned to the user.

The use of partial answers is twofold. First, a partial answer contains an incremental query that can be submitted to the system in order to obtain the complete answer efficiently, once the unavailable data sources are again available. Second, a partial answer contains data from the available sites that can be extracted. To extract data, we use secondary queries, which we call *parachute queries*. A set of parachute queries is associated to each query and can be asked if the complete answer cannot be produced.

In the rest of the section, we discuss an example that illustrates a practical use of partial answers.

1.1 Example

To be more concrete, let us illustrate the problem and our solution with the following example, concerning a hospital information system.

Consider an hospital that consists of three services: administration, surgery, and radiology. Each service manages its own data and is a data source. A mediator system provides doctors with information on patients. The mediator accesses the data sources to answer queries. Figure 1 contains mediator schema; three service data sources are integrated: administration, surgery, and radiology. The schema contains a relation identifying patients (*patient*), a relation associating the local identifiers of patients (*map*), and two relations identifying medical treatments (*surgery* and *radiology*). The data from the *patient*, *surgery* and *radiology* relations are located respectively on the local data sources. The *map* relation is located

id	name	age	address
1	Durand	56	Paris
2	Dupont	38	Versailles
3	Martin	70	Suresnes

(a) patient

patient_id	radiology_id	surgery_id
1	r_1	s_1
2	r_2	s_2
3	r_3	s_3

(b) map

id	date	description
s_1	10/03/96	appendicitis
s_2	23/07/97	broken arm
s_3	30/05/96	appendicitis
s_3	23/10/97	broken leg

(c) surgery

id	date	description	xray
r_2	22/07/97	right arm	$\heartsuit_1$
r_3	23/10/97	right leg	$\heartsuit_2$

(d) radiology

Fig. 1. Hospital Information System Schema. Figure (a) is the *patient* relation from the administrative service. Figure (b) maps the identifiers from the data sources – this relation is contained in the mediator. Figure (c) is the *surgery* relation from the surgery service. Figure (d) is the *radiology* relation from the radiology service. The symbol ♡ represents a digitized x-ray.

in the mediator. (We assume there is no semantic heterogeneity between data sources; this problem is tackled in [4].)

A typical query that a doctor may ask is the following, *select the date, type and description of surgeries and the X-ray for the patient named Martin, that occur on the same date*:

```
Q: select surgery.date, surgery.description,
       radiology.date, radiology.xray
   from patient, surgery, radiology, map
   where patient.name = "Martin" and
       patient.id = map.patient_id and
       map.surgery_id = surgery.id and
       map.radiology_id = radiology.id and
       surgery.date = radiology.date;
```

The answer to this query is:

surgery.date	surgery.description	radiology.date	radiology.xray
23/10/97	broken leg	23/10/97	$\heartsuit_2$

Each service administrates its own data source. In particular, the administrator of the data source in, say the radiology service, can decide at any time to shut down its system to perform maintenance. In the meantime, this data source is unavailable when the mediator tries to access it. Using a classical mediator system, the query described above, which involves radiological data cannot be answered when maintenance is performed in the radiology service. Now, with a mediator providing partial

answers, parachute queries may be associated with the original query. In our example, we could consider two parachute queries: (1) for the case where the radiology source is unavailable and (2) for the case where the surgery source is unavailable.

```
PQ1: select surgery.date, surgery.description
     from patient, surgery, map
     where patient.name = "Martin" and
        patient.id = map.patient_id and
        map.surgery_id = surgery.id;

PQ2: select radiology.date, radiology.xray
     from patient, radiology, map
     where patient.name = "Martin" and
        patient.id = map.patient_id and
        map.radiology_id = radiology.id;
```

Suppose the radiology data source is unavailable when the doctor asks query Q. The complete answer cannot be computed; the system can however obtain data from the administration and surgery data sources. The system returns a partial answer which notifies the doctor that the query cannot be answered because the radiology data source is down. The parachute queries that are associated to the query can be evaluated using the obtained data. The doctor obtains the following answer to the parachute query PQ1;

date	description
30/05/97	appendicitis
23/10/97	broken leg

Using the same obtained data, the system also generates an incremental query that will efficiently compute the complete answer once the radiology data source is available again. The incremental query retrieves data from the radiology data source and reuses data already obtained from the administration and surgery data sources (this incremental query is described in Section 2)

1.2 Contributions

In summary, this paper describes a novel approach to handling unavailable data sources during query processing in heterogeneous distributed databases. We propose a framework for partial answers; an algorithm for the evaluation of queries in presence of unavailable sources; an algorithm for the construction of incremental queries; and an algorithm for

the evaluation of parachute queries. We have implemented part of these algorithms in the Disco prototype [10].

We present in Section 2 an overview of our approach, and in Section 3 our algorithms. We discuss related work in Section 4. We conclude the paper in Section 5 by summarizing our results and discussing future work.

2 Overview of Partial Answers

Let us consider again the initial query described in the introduction. If all sites are available, then the system returns a complete answer. The complete answer is the set of tuples obtained from the root of the execution plan.

Let us suppose now that site radiology is unavailable, while the other sites are available. A complete answer to this query cannot be produced. We propose a solution that, in such a case, does the following:

phase 1 each available site is contacted. Since the radiology site is unavailable neither the join between the *radiology* relation and the data obtained from the administration site, nor the join with the *surgery* relation can be performed. The data from the administration site, i.e. the result of the sub-query SQ1 *select * from patients, treatments where patient.name = "Martin" and patient.id = treatment.patient_id*, and the *surgery* relation, i.e. the result of sub-query SQ2 *select * from surgery*, can however be obtained and materialized on the site where query processing takes place. SQ1 and SQ2 denote data materialized locally in relations R1 and R2.

phase 2 an incremental query, Qi, semantically equivalent to the original query, is constructed using the temporary relations materialized in phase 1 and the relations from the unavailable sites. In our example Qi is:

```
select radiology.date, radiology.description,
       R2.date, R2.description
from R1, R2, radiology
where R1.radiology_id = radiology.id
  and R1.surgery_id = R2.id ;
```

Qi is semantically equivalent to the original query under the assumption that no updates are performed on the remote sites.

A partial answer is returned to the user. It is a handle on the data obtained and materialized in phase 1, as well as on the query constructed

in phase 2. In the next section we propose two algorithms that implement these two phases for the construction of partial answers.

A partial answer can be used in two ways. First, the incremental query Qi, constructed in phase 2, can be submitted to the system in order to obtain the final answer. Evaluating Qi only requires contacting the sites that were unavailable when the original query was evaluated. In the example, R1 and R2 denote data materialized locally, only *radiology* references data located on a remote site.

When Qi is evaluated, the system returns either a complete answer, or another partial answer depending on the availability of the sites that were previously unavailable[1]. When Qi 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 Qi is generally different from the execution plan used for the original query. If the sources that were unavailable during the evaluation of the original query are now available, then a complete answer is returned. Under the assumption that no relevant updates are performed on the remote sites, this answer is the answer to the original query.

Submitting Qi, instead of the original query, in order to obtain the complete answer presents two advantages. A complete answer can be produced even if all the sites are not available simultaneously. It suffices that a site is available during the evaluation of one of the successive partial answers to ensure that the data from this site is used for the complete answer. Moreover, Qi involves temporary relations that are materialized locally; evaluating Qi is usually more efficient than evaluating the original query.

Second, data can be extracted from a partial answer using parachute queries. Parachute queries are associated to the original query; they may be asked in case the complete answer to the original query cannot be produced. In the next section, we propose an initial algorithm for the evaluation of parachute queries.

3 Algorithms

3.1 Architecture

For our algorithms, we consider an architecture that involves an application program, a mediator, wrappers, and data sources. During query

[1] Possibly, successive partial answers are produced before the complete answer can be obtained.

processing, the application program issues a query to the mediator. The mediator transforms the query into some valid execution plan consisting of sub-queries and of a composition query (the algorithms we propose are independent of the execution plan). The mediator then evaluates the execution plan. Evaluation proceeds by issuing sub-queries to the wrappers. Each wrapper that is contacted processes its sub-queries by communicating with the associated data source and returning sub-answers. If all data sources are available, the mediator combines the sub-answers 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 data 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. 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 new operator `submit` contacts a remote site to process a sub-query. During the `open` call to `submit` a network connection to the remote site is opened. In this paper, we assume that if the `open` call to the wrapper succeeds, then the corresponding data source is available and will deliver its sub-answer 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.

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 the incremental query.

We describe a two-step 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 the incremental query from the annotated execution plan. A partial answer

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

Fig. 2. The evaluation algorithm.

is then returned to the user. It is a handle on both the data materialized during the first step and the query constructed in the second step.

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. 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 opens all its children, instead of giving up with the first unavailable child, we implement a simple form of query scrambling [1]. See Figure 2 for an outline of the algorithm.

After the open call 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 in 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

```
construct(execution_plan) returns Incremental Query {
  if available() then {
    return the temporary relation 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
  }
}
```

Fig. 3. Construction of the incremental query.

get-next call and storing the result. The resulting tree is passed to the construct algorithm.

Construct algorithm We construct a declarative query from an annotated execution plan by constructing a declarative expression for each operator in the tree in a bottom-up fashion. The declarative expressions are nested to form the incremental query.

Operators marked available generate a declarative expression that accesses the materialized intermediate result. It is an expression of the form `select` $*$ `from` x `in` r, where x is a new unique variable and r is the name of the temporary relation holding the materialized intermediate result. 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 incremental query, see Figure 3, 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 incremental query, together with the annotated execution plan is used to return a partial answer.

3.3 Extraction Algorithm

We present an algorithm for extracting information from a partial answer, 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 4. First, a query is generated for each intermediate result materialized in the annotated execution plan using the construct algorithm. 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 evaluating the parachute query on one materialized relation. Otherwise, we cannot return any answer to the parachute query. Query containment is defined in [11]. This problem is exactly the same as matching a query against a set of materialized views; an algorithm similar to this one is implemented in ADMS [5].

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

Fig. 4. The extraction algorithm.

An improvement in the evaluation of parachute queries would consist in using a more elaborate evaluation algorithm. We can utilize for this problem, the results of [7] where an algorithm for answering queries using views is proposed. This algorithm would allow to combine several materialized views to evaluate a parachute query.

4 Related Work

Multiplex [9] tackles the issue of unavailable data sources in a multidatabase system and APPROXIMATE [12] tackles the issue of unavailable data in a distributed database. Both systems propose an approach

based on approximate query processing. In presence of unavailable data, the system returns an approximate answer which is defined in terms of subsets and supersets sandwiching the exact answer.

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

References [6] and [8] 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 [8] 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 [8], 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.

5 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 which is a handle on data obtained and materialized from the available sources and on an incremental query that can be used to efficiently obtain the complete answer. Second, relevant information can be extracted from a partial answer using parachute queries. We have implemented our approach [10].

The use of parachute queries provides a very flexible and familiar interface for application programs. However, formulating parachute queries may be a burden for the application programmer. We suspect that relevant parachute queries can be automatically generated given the origi-

nal query. We have started investigating interesting classes of parachute queries and algorithms to generate them [3]; we have also studied performance trade-offs in a system dealing with parachute queries.

Acknowledgments

The authors wish to thank Laurent Amsaleg, Stéphane Bressan, Mike Franklin, Rick Hull, Tamer Oszu and Louiqa Raschid for fruitful discussions, and Mauricio Lopez for comments on previous drafts of this paper.

References

1. 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.
2. Philippe Bonnet and Anthony Tomasic. Partial answers for unavailable data sources. Technical Report RR-3127, INRIA, 1997.
3. Philippe Bonnet and Anthony Tomasic. Parachute queries in the presence of unavailable data sources. Technical Report RR-3429, INRIA, 1998.
4. S. Bressan and C.H. Goh. Answering queries in context. In *Proceedings of the International Conference on Flexible Query Answering Systems, FQAS'98*, Roskilde, Denmark, 1998.
5. 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.
6. T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.
7. 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.
8. A. 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.
9. A. Motro. Multiplex: A formal model for multidatabases and its implementation. Technical Report ISSE-TR-95-103, George Mason University, 1995.
10. A. Tomasic, R. Amouroux, Ph. 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.
11. Jeffrey D. Ullman. *Principals of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.
12. 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.