Leveraging Mediator Cost Models with Heterogeneous Data Sources

Hubert Naacke, Georges Gardarin, Anthony Tomasic

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Leveraging Mediator Cost Models with Heterogeneous Data Sources

Hubert Naacke , Georges Gardarin , Anthony Tomasic

N ° 3143

Mars 1997

THÈME 1

ISSN 0249-6399

INRIA
ROCQUENCOURT

Leveraging Mediator Cost Models with Heterogeneous Data Sources

Hubert Naacke* , Georges Gardarin* , Anthony Tomasic*

Thème 1 — Réseaux et systèmes
Projet Rodin

Rapport de recherche n ° 3143 — Mars 1997 — 17 pages

Abstract: Distributed systems require declarative access to diverse data sources of information. One approach to solving this heterogeneous distributed database problem is based on mediator architectures. In these architectures, mediators accept queries from users, process them with respect to wrappers, and return answers. Wrapper provide access to underlying data sources. To efficiently process queries, the mediator must optimize the plan used for processing the query. In classical databases, cost-estimate based query optimization is an effective method for optimization. In a heterogeneous distributed databases, cost-estimate based query optimization is difficult to achieve because the underlying data sources do not export cost information. This paper describes a new method that permits the wrapper programmer to export cost estimates (cost estimate formulas and statistics). For the wrapper programmer to describe all cost estimates may be impossible due to lack of information or burdensome due to the amount of information. We ease this responsibility of the wrapper programmer by leveraging the generic cost model of the mediator with specific cost estimates from the wrappers. This paper describes the mediator architecture, the language for specifying cost estimates, the algorithm for the blending of cost estimates during query optimization, and experimental results based on a combination of analytical formulas and real measurements of an object database system.

Key-words: Heterogeneous Distributed Database, Mediator, Wrapper, Optimization, Cost Model

(Résumé : tsvp)

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

* e-mail: FirstName.LastName@inria.fr

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

Modèles de coûts pour des sources de données hétérogènes

Résumé : Les systèmes distribués accèdent à des sources d'informations diverses au moyen de requêtes déclaratives. Une solution pour résoudre les problèmes liés à l'hétérogénéité des sources repose sur une architecture à base de médiateurs. Dans cette architecture, le médiateur accepte en entrée une requête de l'utilisateur, la traite à l'aide d'adaptateurs et renvoie la réponse. Les adaptateurs fournissent l'accès aux sources de données. Pour traiter une requête de manière efficace, le médiateur doit optimiser le plan décrivant le traitement de la requête. L'optimisation de requêtes basées sur l'estimation du coût est bien adaptée aux bases de données classiques. Toutefois, pour les bases de données hétérogènes, cette optimisation basée sur le coût est difficile à mettre en place car les sources de données n'exportent pas d'information de coût. Cet article décrit une nouvelle méthode permettant au développeur d'adaptateur d'exporter des estimations de coût (formules de coût et statistiques). Le développeur d'adaptateur ne peut cependant pas décrire l'ensemble des estimations de coût par manque d'information ou si cette description est trop vaste. Nous allégeons la charge du développeur en complétant sa description avec le modèle de coût générique prédéfini du médiateur. Cet article décrit l'architecture du médiateur, le langage pour spécifier les estimations de coût, l'algorithme pour mélanger les informations de coûts pendant l'optimisation et des résultats expérimentaux basés sur une combinaison de formules analytiques et de mesures effectuées sur un système de bases de données objet.

Mots-clé : Base de Données Hétérogène Distribuée, Médiateur, Adaptateur, Optimisation, Modèle de coût

1 Introduction

Declarative access to diverse data sources of information is recognized as a key issue in heterogeneous systems. The concept of a *mediator* [Wie93] has been proposed as a good basis for giving integrated views of multiple heterogeneous data sources. Declarative queries upon the views have to be processed efficiently by the mediator. The work described in this paper is part of the DISCO [TRV96] project at INRIA. DISCO has a mediator based architecture for accessing heterogeneous distributed databases. The architecture consists of *data sources* that provide raw data, *wrappers* that provide interfaces to data sources, *mediators* that provide declarative query access to multiple wrappers, and *clients* that provide queries to mediators and accept answers returned from mediators. Several projects follow a similar architecture (TSIMMIS [GMHI+94], Garlic [C+95], DIOM [LP95], Information Manifold [KLSS95], HERMES [SAB+97], COIN [GMS94], IRO-DB [G+95]) or related architectural frameworks (Infosleuth [R+96]).

Declarative access in the form of queries on data sources gives a degree of freedom to the mediator to determine the best plan for the execution of the query. From a declarative query, the mediator can generate multiple access plans involving local operations at the data source level and global ones at the mediator level. The plans can differ widely in execution time due to varying local processing costs, communication costs, and mediator processing costs. The method for choosing the best plan remains an open issue. In classical database systems, the query *optimizer* generally implements a search strategy using a cost model. Plans are generated and compared using a cost estimate derived from database statistics and cost formulas to compute the cost of each operator of the plan. This approach cannot easily be applied to heterogeneous databases with multiple data sources because: (i) data sources do not report needed statistical information (e.g., HTML files, object-oriented databases); (ii) cost formulas for processing an operator (e.g., selection, or join) vary radically depending on the implementation of the wrapper and the underlying data source; (iii) communication costs are difficult to determine and may vary over time according to the network or system loads (e.g., on the Internet).

Various solutions to the cost estimate problem have been proposed in the past [BGW+81, AHY83, YC84]. Recently, the calibration approach was introduced in [DKS92] and extended to object systems in [GST96]. A calibrating procedure is proposed that estimates the coefficients of a generic cost model, which can be specialized for a class of systems. This approach has been implemented in Pegasus [SAD+95] and in the IRO-DB project [GGT95]. The main problem for calibration appears when a data source does not follow the generic cost model of these systems (which cannot be changed). We believe that this situation arises frequently in a heterogeneous environment. Another approach, proposed in the HERMES [ACPS96] project, records the cost information for every query issued to a data source. Cost estimates for new queries are based on the history of queries issued to a data source. Although very interesting for uniformly used sources, the approach is limited for data sources which are queried with dissimilar predicates or which are rarely queried. We survey in more detail the existing proposals in Section 6.

In this paper, we describe a new approach to the problem of evaluating the cost of a query plan in a heterogeneous DBMS with multiple data sources. The approach relies on combining a generic cost model with specific cost information exported by wrappers. The wrapper implementor specifies any part of the cost information of the data source, from nothing to everything. By default, the mediator implements its own generic cost model, and when possible, corrects it with the information imported from the wrappers. Thus, the generic cost model is used by the mediator for unknown data source operations, while the wrapper cost models provide, through a standard interface, more accurate cost formulas. The proposed heterogeneous and extensible cost model is currently being implemented in DISCO [TRV96]. In the validation section of this paper, we provide evidence of the benefits of this new approach.

To support the extensible cost model, we provide a *tool* for the wrapper implementor to export *statistics, size and cost computation rules*. The statistics reflect properties of the underlying data source such as the cardinality of a collection. The size rules reflect the change in result sizes due to an operation, such as the reduction in cardinality due to a select operation. The cost rules compute cost estimates, such as the estimated response time for a scan operation. Specific cost information are imported from a wrapper to the mediator when a data source is registered. Then, during query processing, some standard cost computation functions of the mediator are overridden by the imported cost functions for the given data source.

For statistics, the wrapper may export a triplet for each collection giving the number of objects, the total size of the collection in term of disk space, the average size of objects, and a triplet for each attribute giving the minimum, maximum, and the number of distinct values. In addition, the presence of indices may be exported. For cost rules, several formulas for each wrapper operation may be exported. One formula computes response time for the first tuple, a second computes response time for the next tuple, and the third computes total work

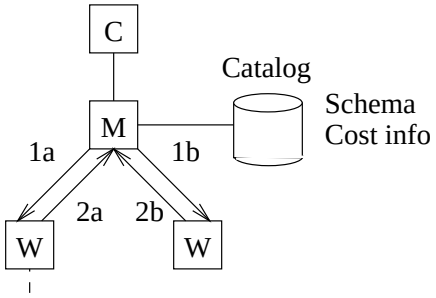

Figure 1: The registration phase of system interaction.

in terms of milliseconds. Size rules are integrated within the cost rules, and may include for each operation formulas to compute the new cardinality and the new total size.

For a simple, predictable data source the wrapper implementor can easily provide accurate cost estimates. (The only difficulty is estimating network performance, a problem which we do not consider in this paper). But for more complicated sources, the burden on the wrapper implementor becomes too large, since the amount of cost information required increases dramatically. In that case, partial information, such as typical cost of frequent operations or even cost of test queries can be exported, thus permitting graceful improvement of the mediator generic cost model.

In summary, we provide an elegant framework for integrating heterogeneous cost models within a generic model. Furthermore, the framework is extensible in the sense that some wrappers may only provide partial information about underlying costs, while others may provide specific information for given queries. The two extreme indeed encompass calibration (i.e., no specific rules for a data source) and historical query caching (i.e., specific information for past queries). Our algorithm to blend rules and statistics from wrappers with the default cost model of the mediators benefits from object-orientation to gracefully extend the generic cost model through overriding statistics and cost computation methods.

The paper is organized as follows. In Section 2, we present an overview of the architecture of the project, focusing on the mediator and wrapper capabilities. We particularly detail the query optimization process and show the importance of the cost model. Section 3 describes how a wrapper provides the necessary statistics, size and cost computation rules. We provide a language for expressing this information. Section 4 gives a detailed overview of the cost computation module in the mediator. The mediator dynamically loads cost information from the wrapper. The cost computation module uses object-orientation concepts such as overriding to blend wrapper information. Section 5 validates the approach through an experimental study using the 007 benchmark on top of ObjectStore. It demonstrates the benefits of our proposal compared to the classical calibrating approach. Section 6 compares our approach with related work. The conclusion in Section 7 summarizes the paper and sketches future topics of research.

2 Architecture and Generic Cost Model

Several projects follow the same general mediator architectural framework as mentioned in the introduction. In this section we describe the architecture of DISCO [TRV96] by describing an overview of the phases and steps required to process a query. We also introduce the generic cost model of the mediator. To simplify the presentation we have suppressed some details.

2.1 Registration Phase

Interaction between the wrapper and mediator occurs in two phases, the *registration* phase and the *query processing* phase. During the registration phase, mediators contact wrappers and upload all the information required to use the wrapper, including cost information. Figure 1 shows a diagram of the steps of interaction between the mediator (M) and wrappers (W) during the registration phase. In this phase, the mediator (in response to instructions from the mediator administrator) calls the wrapper (Step 1a and 1b). The wrapper returns a collection of information needed for query processing (Step 2a and 2b). The collection contains the *schema* of the wrapper (reflecting the schemas of the underlying data sources, not shown here), *capabilities* of the wrapper (the set of operations the wrapper can execute), and *cost* information. Schema and cost information are

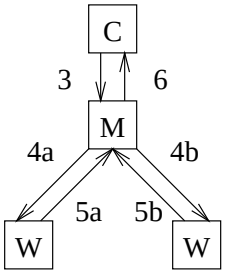

Figure 2: The query phase of system interaction.

stored in the mediator catalog. (The nature of the schema and capabilities information and its integration into query processing is considered elsewhere [KTV97]. In this paper we will assume that all wrappers can execute all operations.) We envision an administrative interface for both the mediator and wrapper to re-register wrappers. This interface is necessary when the cost formulas are improved by the wrapper implementor, or the statistics become out of date.

2.2 Query Processing Phase

In the second phase, queries are processed. The second phase typically happens multiple times for every registration phase. Figure 2 shows a diagram of the steps involved. In Step 3 the client issues a query to the mediator and waits for an answer. The mediator accepts the query and decomposes it into subqueries, one for each wrapper, and a composition subquery. In Steps 4a and 4b, the mediator issues the subqueries to the wrappers and waits for a response. The wrappers process the subqueries by consulting the associated data sources (not shown) and generate subanswers that are returned to the mediator in Steps 5a and 5b. The mediator combines the subanswers by using the composition subquery and generates the final answer that is returned to the client in Step 6. Note that in DISCO, the query in Step 3 is *declarative*, written in simple object/relational SQL language. The subqueries of Step 4 are *algebraic* and extend the relational logical operators.

To accomplish the translation from Step 3 to Step 4, the mediator does several things. It *parses* the client query, it *transforms* the query, written with respect to a global view, into a query over local schemas, and it *optimizes* the query to produce the best query execution *plan*. The mediator then executes the best plan, resulting in Step 4 and the subsequent steps. During optimization, the mediator estimates the cost of various operations and of entire plans. The mediator chooses the most specific information available as the result of registering wrappers. As discussed in the introduction, the best plan depends on good cost estimates for the subqueries sent to the wrapper. The mechanism described in this paper results in good cost estimates.

The mediator constructs several plans for the optimization of a query. A plan consists of a tree of algebraic operators. Although there exist many different data source managers, the basic algebraic operators are always the same; typically they include all operators of a classical object algebra [ABD+92, AK93]. Thus, the mediator algebra covers the following common operators:

- unary operators including `scan`, `select`, `project`, `sort`;
- binary operators including `join`, `union`;
- aggregate operators for elimination of duplicates or computing aggregate functions (e.g., `sum` and `average`), plus
- an operator `submit` that is used to model the issuing of a subplan to a wrapper.

2.3 Mediator generic cost model

The role of the mediator optimizer is to select the most efficient plan among the alternatives based on the cost estimations. When no specific information are given by wrappers, the mediator estimates the cost of plans using a cost model. There are several major components of the cost : CPU cost, IO cost, and Communication cost. However, for simplicity, the generic cost model does not separate CPU and IO costs, which are buried in global cost formulas parameters.

The cost model depends on time parameters and statistical parameters. We assume in this paper a uniform communication cost; discrepancy of communication costs is a subject of future research. Time parameters

come in three forms: the overhead required to start processing *TimeFirst*, the time required to deliver each tuple *TimeNext*, and the time to get all tuples *TotalTime*. *TimeFirst* accounts for query start up time and, in particular, `sort` operations. *TimeNext* gives the average time cost of each tuple. The time is measured in milliseconds.

For unary operators, the generic cost model of the mediator considers two cases: sequential scan, and index scan. The cost formulas are established using a calibrating approach [GST96]. These formulas requires the selectivity of a selection that can be derived from the minimum, maximum, and number of distinct values of the restricted attributes. Furthermore, to be able to select the relevant formula, the data source must export the presence of indexes on attributes. In the generic cost model, clustering is not considered.

For binary operations, the generic cost model of the mediator considers three cases : index join, nested loops and sort merge join. The formulas are those of [GST96]. When an index is existing, the index join formula is selected, otherwise the best of the two others is chosen. Applying these formulas does not require more information that those define above for selections, as the join cardinality can be estimated as $1/Min(CountDistinct(A), CountDistinct(B))$. Thus, no further statistics are required for the generic cost model. The same is true for aggregate computation.

2.4 Implementation

The implementation of DISCO uses Java as the common language for mediator processing and wrapper processing. The cost formulas exported by wrappers are implemented as code generated from a compiler of the cost formula language. The resulting code is shipped to the mediator during the registration phase.

Encapsulating cost functions via code-shipping yields fast evaluation time for the functions during query optimization. Fast evaluation times are a requirement due to the computational intensity of query optimization. In addition, since cost formulas are shipped during the registration phase, the loading of cost formulas does not delay query processing. Finally, since the shipped code executes in the process space of the mediator, the entire library of code in the mediator (including the standard Java library) is available to the wrapper implementor when the cost formulas are defined.

3 Cost Communication Language

Local wrappers export data and operations described by the source administrator using a common object model. To base the system on solid foundations, we selected a subset of CORBA Interface Definition Language (IDL) [OMG95] to specify data source interfaces. To export statistics of collections, including cardinality, selectivity, object size, etc., we extend the *interface* body with a *cardinality* section. To overcome the limitations of using generic cost formulas in the mediator query optimizer, we also add a *cost formula* section which provides specific formula to the mediator. The *cost formula* section aims to better calculate the cost of an algebraic operation, i.e., a node in the query tree. In this section we describe the interface between the wrapper and the mediator. The IDL interface is extended in order to export statistics and cost information.

3.1 Exporting Interfaces

To define the objects exported by each data source, we use a subset of IDL. It allows data source providers to easily map the interface and structure of the objects they provide, in a definition language close to existing standards [Cat95, OMG95]. For data sources conforming to these standards, the mapping is quite straightforward. Interface definitions include typed attributes, operations, and exceptions.

Interface declaration may also include constant and type declarations. Support of inheritance and aggregation of interfaces is planed. Elementary types are built in and complex types can be constructed using the tuple and sequence constructors. Like in IDL, relationships are not supported, but we believe that it is not fundamental as local join operations can hide them. Figure 3 gives an example of a simple interface description. We define the employee interface with typical attributes and operations.

3.2 Exporting Statistics

The local data sources also export statistics together with interfaces. Statistics are used as parameters in the mediator cost model formulas. Exported statistics are simple, they describe data sources collections in the same way as in former calibrating approaches [DKS92, GST96]. The wrapper implementor expresses the statistical

```
interface Employee {
  attribute Long salary;
  attribute String Name;
  short age();
}
```

Figure 3: An example interface.

properties of a collection through two special methods attached to each interface description. To distinguish theses two methods from other possible ones, we add the keyword `cardinality` in front of the signatures of the both methods. The first method, named `extent` returns the number *CountObject* of objects in the extent, the size *TotalSize* of the extent in bytes, and the average size *ObjectSize* of an object in bytes. The second method, named `attribute` describes, for a given attribute *AttributeName*, a boolean *Indexed* indicating the existence of an index, the number *CountDistinct* of distinct values for the attribute in the extent, and the minimum *Min* and maximum *Max* values for the attribute. Since the minimum and maximum values may be of various types, we encode this object in a special polymorphic `Constant` object. Figure 4 shows the two cardinality methods added into the interface definition of employee.

```
interface Employee {
  ...
  cardinality
    extent(out long CountObject, out long TotalSize, out long ObjectSize);
  cardinality
    attribute(in String AttributeName, out Boolean Indexed,
              out Long CountDistinct, out Constant Min, out Constant Max);
}
```

Figure 4: An example interface extended with statistics.

More precisely, we extend the *interface_body* BNF syntax (see [OMG95]) to include the `cardinality` methods, as shown in Figure 5.

The mediator calls the two methods extent and attribute during the interface registration, and stores the interface statistics in its catalog.

This cardinality section is purely descriptive and it provides enough information to map this IDL interface into a programming language. We show in Figure 6 the *cardinality* implementation for the `employee` interface in Java. These functions are very simple in the example.

3.3 Exporting formulas

As explained in the previous section, we assume that each data source wrapper is able to provide a basic object algebra. This is in general not true, but relaxing this assumption is out of the scope of this paper.

```
<interface_body> ::= <export>*
<export>         ::= <type_dcl>;
                   | <const_dcl>; | <except_dcl>; | <attr_dcl>; | <op_dcl>;
                   | <card_dcl>;
<card_dcl>       ::= cardinality <extent_sign>; | cardinality <attribute_sign>;
<extent_sign>    ::= extent(out long CountObject, out long TotalSize, out long ObjectSize)
<attribute_sign> ::= attribute(in String AttributeName, out Boolean Indexed,
                     out Long CountDistinct, out Constant Min, out Constant Max)
```

Figure 5: BNF for cardinality descriptions

```
void extent (LongHolder CountObject, LongHolder TotalSize, LongHolder ObjectSize) {
   CountObject.value = 10000;
   TotalSize.value = 15;
   ObjectSize.value = 120;
}

void attribute(String AttributeName, BooleanHolder Indexed,
         LongHolder CountDistinct, ConstantHolder Min, ConstantHolder Max) {
   if (AttributeName.equals ("salary")) {
      Indexed.value = true; CountDistinct.value = 10000;
      Min.value = 1000; Max.value = 30000;
   }
   else if (AttributeName.equals ("Name")) {
      Indexed.value = true; CountDistinct.value = 10000;
      Min.value = "Adiba"; Max.value = "Valduriez";
   }
}
```

Figure 6: The Employee cardinality method definitions.

Cardinality of collection C	C.`CountObject`
Size of collection C in bytes	C.`TotalSize`
Average object size in C	C.`ObjectSize`
Presence of an index for an attribute A	$C.A$.`Indexed`
Count of distinct values for an attribute A	$C.A$.`CountDistinct`
Minimum value of an attribute A	$C.A$.`Min`
Maximum value of an attribute A	$C.A$.`Max`

Figure 7: Name scheme for statistics in a formula.

However, even if implementing standard operators, a local data source may implement it in a very specific way, e.g., using a bit map index, a pointer chasing operator, or an efficient clustering algorithm. Thus, the generic cost model of the mediator optimizer will not be valid for this data source. To overcome this difficulty, we extend the interface definition with an optional new section to give cost formulas that will override the generic cost model of the mediator.

A *cost formula* is either collection oriented or operator oriented. If the cost formula is included inside the interface definition of a particular collection, it describes cost functions for operators on that particular collection. If the cost formula is apart from any interface definition, it describes cost functions that are not specially related to a particular collection but rather to an operator, i.e., the cost formula is valid for all collections of the source that have no collection specific cost formula. We will focus on the second kind of cost formulas; the first kind can be expressed using the same interface by naming explicitly the collection.

3.3.1 Cost Formula Syntax

Cost formulas have the standard mathematical syntax (c.f., the BNF grammar in Figure 9). Wrapper writers may use all the statistics, from the collection interfaces, by simply naming them. The naming convention is based on path expressions, such as *Collection.Attribute.Statistic*, where *Collection* is a collection name, *Attribute* is an attribute of the collection, and *Statistic* is a term referring to a statistic. *Attribute* and *Collection* may be omitted in non-ambiguous cases. Figure 7 lists the variable names for statistics that can be used in a formula.

In addition, wrapper implementors may define their own local variables or functions to parameterize their formulas. For example, they may define the variable PageSize = 4000. They may also invoke functions from the standard Java library. For example, a *cost formulas* could depend on the current date D accessible by the Java call: $D = System.currentTimeMillis()$.

To better introduce the cost formula syntax, we give an example of a formula for a linear scan on the collection Employee :

$$\texttt{TotalTime} = 120 + \texttt{Employee.TotalSize} * 12 + \frac{\texttt{Employee.CountObject}}{\texttt{Employee.CountDistinct}}$$

```
scan(employee) ⇐
   TotalTime    = 120 + Employee.TotalSize * 12 + Employee.CountObject / Employee.CountDistinct

select(C, A = V ) ⇐
   CountObject  = C.CountObject * selectivity(A,V)
   TotalSize    = CountObject * C.ObjectSize
   TotalTime    = C.TotalTime + C.TotalSize * 25
```

Figure 8: Two example rules for computing both time and statistic formulas.

3.3.2 Operator-Formula Attachment

In addition to writing formulas, the wrapper implementors indicate the operator on which a formula apply. We use a rule-based approach to bind each formula with its associated operator. We describe in the next section how the mediator matches such rules.

Each rule describes the cost for one operator. A rule is divided into a head and a body: (i) The rule head represents the operator and its arguments ; an argument may be bound to a collection or a predicate, or may be a free variable. (ii) The rule body is the formula itself ; the body may contain more than one formula depending on how many costs are provided by the wrapper implementor.

In Figure 8, we show some example rules for a `scan` operation on the collection employee and a `select` operation. In the figure, `employee` refers to the employee collection, A is a free variable that will be bound to a particular attribute name, V is a free variable that will be bound to a particular value, and `selectivity(`A`,` V`)` refers to an ad-hoc function defined by the wrapper implementor, that could handle, for example, histogram statistics [IP95, PIHS96]. Variables without a collection name refer to the result of the formula.

Given the plan `select(scan(employee), salary = 10)`, both of the rules match a part of the plan. The first rule matches `scan(employee)`, invoking the computation of `TotalTime` in the first rule. The second rule matches `select(c, salary = 10)`, where c represents the result of the `scan` and matches C, and A matches `salary` and V matches 10, invoking the three computations of the formulas. The last computation uses the previous `TotalTime` result to compute a new `TotalTime` result. Note that for both rules, several formula are missing. Default formulas (i.e., that of the generic cost model) are used in this case.

The rule approach provides a very large advantage to the wrapper implementor. The presence of free variables in the rule head makes very easy to adjust the cost precision by writing several rules, each rule more and more specific. However, the drawback to this expressiveness is the proliferation of query-specific cost rules that tends to slow down the cost estimate process. In other words the cost rules overriding mechanism should not induce significant workload on the mediator site. That is why we do not use the standard overriding mechanism of Java, but implement our own efficient one based on kind of virtual tables.

As we mentioned above, both statistical formulas and cost formulas are used to estimate the cost of a plan (tree). The cost of the execution of the plan is determined with a two step bottom-up algorithm described in the next section. In the first step, each operator submitted to a remote data source is matched against the rule head patterns. If the operator name match the rule head, the binding mechanism unifies each variable in the pattern with a corresponding value from the operator being estimated. Therefore, two rules may have different matching levels: (i) unification on the collection name; (ii) unification on the attribute name; (iii) unification on the predicate operation and the predicate arguments. In this case, we select the most specific rule, with more bound parameters. In case of multiple rules matching at the same level, we select the first one in the order given by the wrapper implementor.

4 Cost Evaluation Algorithm

In this section we describe the cost evaluation algorithm. The algorithm executes in the mediator as part of query processing. Before query processing begins, wrappers are registered with the mediator. During registration, wrapper rules are *integrated* into the mediator cost model. During query processing, the integrated rules are used to estimate the cost of query execution plans.

```
< cost_rule >   ::= < operator > ⇐ < formula >*
< operator >    ::= scan( < collection > )
                  | select( < collection >, < sel_pred > )
                  | project( < collection >, < proj_pred > )
                  | join( < collection >, < collection >, < join_pred > )
< collection >  ::= <name> | <variable>
< sel_pred >    ::= < attribute > = < value >
< proj_pred >   ::= < attribute > , < attribute >*
< join_pred >   ::= < attribute > = < attribute >
< attribute >   ::= <name>
< value >       ::= <number> | <string>
< formula >     ::=< result > = < math_expr > ;
< math_expr >   ::= parameter
                  | <number>
                  | < math_expr > < op > < math_expr >
                  | < function > ( < math_expr > ,< math_expr >*)
< op >          ::= + | - | * | /
< function >    ::= <name>
< result >      ::= TotalTime | TimeFirst | TimeNext
                  | CountObject | TotalSize
```

Figure 9: Grammar for cost rules description

4.1 Cost Formula Integration

Integration consists of compiling the rules written by the wrapper implementor and transmitting the results of compilation to the mediator. Usually, these two steps are decoupled as a convenience to the wrapper implementor. Additional variables and functions, that come with the wrapper cost formula, are also stored in the mediator.

In compiling a rule, the head of each rule is converted into an internal structure that represents the operator pattern, e.g., select(C, A=V). The rule head internal structure is similar to that of an operator in the query tree. The rule body is converted into object code. This compilation speeds up both the subsequent matching between query tree operators and rule heads and the evaluation for cost formula. The rules are grouped into three *scopes* based on their applicability domain: wrapper-scope, collection-scope and predicate-scope (see Figure10). Wrapper-scope rules apply to any collection and any predicate of the source. Collection-scope rules apply to a specific collection with any predicate. Predicate-scope rules have the most restricted domain; they apply only to a specific collection with a specific predicate. This grouping of rules into scopes forms a specialization hierarchy.

Furthermore, the mediator has two additional scopes, the default-scope and the local-scope. The local-scope is similar to a wrapper-scope but applies to operators local to the mediator[1]. The default-scope encapsulates all other scopes and contains a rule for all variables and operators. If a more specific rule is not found, the default-scope rule is used. As an elegant consequence, we are able to use the specialization and matching mechanism across all operations and scopes.

4.2 Cost Estimation

Once the rule integration is done, the query processor in the mediator can generate cost estimates for a plan. A plan consists of a tree of operator nodes. To estimate the cost of a plan, a traversal of the plan is done. This recursive tree traversal has two phases: a top-down traversal from the root to the leaves and then a bottom-up traversal from the leaves to the root. During the first phase cost formulas are associated with nodes. During the second phase the cost of each operator is computed. Since the first phase is top down, each node has cost formulas associated with it before the costs are computed. Since the second phase is bottom-up, the cost of the children of an operator are computed before the cost of an operator are computed. This algorithm is shown in Figure 11. We describe in detail the 3 steps of the algorithm.

[1]The rules for the local-scope are different than other scopes because the mediator processes local operators using a physical algebra instead of a logical algebra.

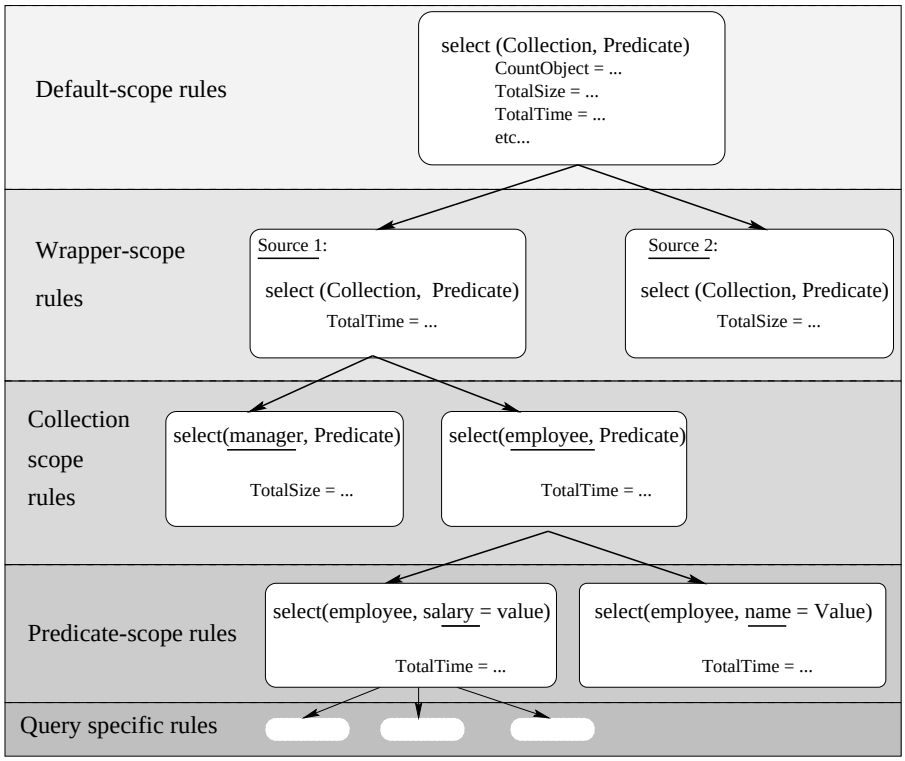

Figure 10: The hierarchic cost formula tree.

```
cost(node) {
1.   associate cost formulas with node
     foreach child in node.children {
2.       cost(child)
     }
3.   apply formulas to node
}
```

Figure 11: Cost Estimation Algorithm

Step 1. Associate cost formulas with node.
This step selects, for each statistic and cost, the most specific rule among all the possible rules that apply for the node. Selecting a rule consists of matching the *node* against the heads of cost rules. Since cost rules are classified, *node* is matched over the scope hierarchy, until the most specialized rule is found.

The matching order is: (1) predicate-scope (names and attributes), (2) operator-scope, (3) collection-scope, (4) wrapper-scope, (5) default-scope. Thus, if no formula have been imported from wrapper, a default-scope rule is selected.

For example, the following operators are ordered by matching order:

```
select(R, P) > select(Employee, P) > select(Employee, salary = A)
select(Employee, salary = A) > select(Employee, salary = 77)
join(R1, R2, P) > join(Employee, Book, P) > join(Employee, Book, x1.id = x2.id)
```

All rules at the same matching level that match a *node* are associated with the node. In addition, several formula may compute different values for the same variable for a node. During the second phase, conflicts between multiple formulas for the same value are resolved by evaluating all formula and choosing the lowest value (see Step 3).

Furthermore, for each selected formula, the list of statistics involved in the formula computation is filled and propagated to children. Then, each child receives the list of statistics they have to compute. For each statistic, a formula has to be found in the scope hierarchy. The list mechanism guarantees the liveness of step 3.

Since every variable is considered, if the matching rule only provides some of the required formula (e.g., the join rule computes only TotalTime), the scope hierarchy is scanned until the first less-specific rule is found (e.g., CountObject and TotalSize are processed by a default-scope formula). The mediator default cost model guarantees that a least one formula is found for every variable for every node.

At the moment, we do not cope with ambiguous matching within a scope. We are currently investigating the significance of a sorting criteria to classify such cases. For instance, for an operator `join(Employee, Manager, predicate)`, both patterns `join(Employee, R2, P)` and `join(R1, Book, P)` match.

Phase 1 of the algorithm can be optimized during this step in the following two ways: (i) at each node the *required* variables are analyzed, depending of the form of each formula. The set of required variables are passed to each child in Step 2. Only formula that compute required variables are associated with a node (and thus, subsequently selected for invocation). (ii) If no variables required from a child node, the recursive call to the child is cut. The savings from this optimization depend on the form of the formulas. In the best case, the root node has formulas containing only constants and consequently no recursive traversal of the tree is performed.

Step 2: Recursive traversal via depth first fetch.
Each child node is recursively called to assign cost formula and compute costs associated with the node. If the above optimization is used, no call is issued to a child if no values are required from the child node.

Step 3: Apply formulas to node.
The associated formulas are invoked and the corresponding variable is assigned a value. This value will be accessed by a parent node during the invocation of its formula. In the case where many formula have been selected to compute the same variable, all formulas are invoked and the lowest value is assigned to the variable.

4.3 Extension to Dynamic Cost Formulas

We present in this section possible extensions to the algorithm. We will implement these extensions in a more general framework. The first extension deals with cost formula adjustment based on historical cost. The second extension uses the *best current estimation* to avoid useless computation, when a previously computed plan is cheaper than the plan being estimated.

4.3.1 Historical Costs

A simple way to have very accurate cost is to extend the scope hierarchy with a *query* scope. In the query scope, specific rules match a wrapper subquery exactly. A new formula is added after a subquery has been executed and the associated formula are now real costs, not estimates. This solution is close to the HERMES [ACPS96] approach based on historical costs. As in this system, we assume that a formula based on real costs is an accurate estimate for the next identical subquery. That is, two executions of the same subquery have the same cost regardless of differences in time. However this solution is restricted, in that new formulas are restricted to one specific subquery and cannot be reused for another, closely related subqueries. (For instance, subqueries that vary only by the constant used a predicate.) We plan to investigate how to *modify* existing formulas depending on the history of query execution costs instead of storing new formulas. One solution takes existing formulas and adjusts the input parameters until the formula returns a cost close to real execution the cost. Thus, we store only the adjusted parameters instead of new formulas. We expect that all formula using the same input parameters can be simultaneously adjusted.

This technique solves a second problem of the HERMES approach, the proliferation of statistical information, in a different way than in HERMES. Instead of constructing summaries of statistical information, we encode the history of the execution in the adjustments to the input parameters.

4.3.2 Avoiding Useless Computation

Recall that optimizer generates several plans. The estimation algorithm provides plan costs in order to make the optimizer to choose the cheapest one. Thus, once a plan has been estimated, any more expensive plan will be rejected. We would like to stop the estimation of a plan in the middle of the process, as soon as the currently computed (sub) cost is greater than the cost of the current best plan. To do this we add a test to Step 3 of the algorithm in Figure 11. The test compares the current local subcost with the cost of the best current plan: if the local subcost is greater than the best current cost, the plan is immediately rejected. We plan to investigate the impact of this heuristic on optimizer performance.

5 Results

We have implemented a version of the algorithm of Section 4 using a logic programming language. This implementation allowed us understand in detail the specialization hierarchy and its implications in determining which formula are used for a given operator. We are currently implementing this algorithm in DISCO. In addition, we studied the impact of improving cost estimates.

To study this impact, we used data from an experimental study of calibrating the costs of access to an ObjectStore database. The queries used in the validation come from the 007 benchmark [CDN93]. The data for the experimental study comes from [GST96].

The goal of [GST96] focused on tuning the cost model of an optimizer by a calibrating approach. First, several invariant coefficients appearing in cost formulas are isolated. Then, a set of queries on a calibrating database on each local site are run to deduce cost formula coefficients. Once the coefficients are set, the OO7 benchmark was run to validate that real execution time are closely estimated by the calibrated formulas.

Let us focus on the index scan experiment of this work. A collection of small objects (called AtomicParts in the OO7 benchmark) is scanned by an index scan. The size of one AtomicPart object is 56 bytes, the collection cardinality is 70000 and its size is 1000 pages. The page fill factor is 96% of 4096 bytes. Figure 12 gives the response time to scan the AtomicParts collection using an index on the attribute *Id*. The distribution of the *Id* value is uniform. On the horizontal axis, the varying parameter is the operator selectivity in the range of [0, 0.7].

The calibrated formula, associated with the index scan operator, has a linear response time estimate. The formula assumes that the number of pages fetched is proportional to the selectivity of the operator. However, the figure shows that real response time is not linear. Thus, we could have a more accurate estimate based on Yao formula [Yao77] that gives the percentage of pages fetched when processing an index scan on a collection: $Yao(sel) = 1 - exp(-1 * (sel * CountObject/CountPage))$; where $CountObject$ is the total number of objects of the collection, and $CountPage$ is the total number of pages of the collection.

Thus a better cost estimate formula for the index scan operator is:

$$cost = IO * CountPage * (1 - exp(-1 * (sel * \frac{CountObject}{CountPage}))) + (sel * CountObject * Output)$$

where the cost to read one page is $IO = 0.025\ s$, the time to process one object is $OutPut = 0.009\ s$, $CountObject = 70000$, and $CountPage = 1000$. Figure 12 shows that the new estimated curve better fits the experimental curve.

Using our framework, this new formula can be expressed by wrapper implementors and easily imported inside the mediator. The wrapper implementor simply provides the rule shown in Figure 13.

6 Related Work

From the early work on distributed query processing [BGW+81, AHY83, DSH+82], it is assumed that the following parameters are available in the system catalog [Cha82, YC84]. For each collection C, the number of objects $|C|$ and average the size of objects s has to be known ; for counting the number of I/O, it is better to be able to derive in some way the number of pages $||C||$. For each attribute, the number of possible values is sufficient to derive the selectivity on equal restrictions. To infer the selectivity of greater than or less than restrictions, the system must know the maximum or minimum values of the constrained attribute. Such parameters are handled by commercial systems, and we keep the same approach. We just defined the standard methods to get these parameters from either the mediator or the wrappers. In case they are not provided, standard values are given, as usual.

To query heterogeneous sources, most modern multidatabase systems assume a way to estimate the cost of a plan using a formula as follows:

TotalCost = local processing cost + communication cost + cost of sub-queries.

The Garlic authors [C+95] mentions that local processing costs of wrappers and their data sources must be estimated by cost models defined by each wrappers individually because there is no universal, generic cost model that is valid for all wrappers and all data sources. We fully agree with this claim, and implement a sophisticated way to blend the various cost models in DISCO. Further, Garlic assumes a calibrating approach for different classes of cost models, which is difficult to implement with numerous and evolving data sources.

The calibrating approach was first introduced in [DKS92]. A logical cost model with cost coefficients was implemented for relational systems in Pegasus [DS95]. The coefficients represent on average how much CPU

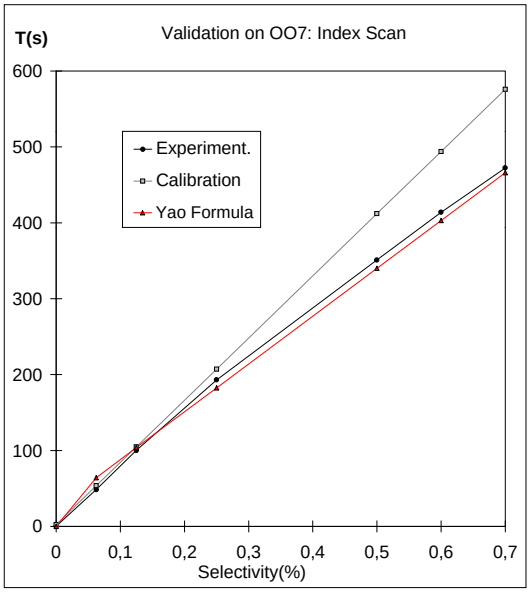

Figure 12: Improvement of ObjectStore calibration.

$$
\begin{aligned}
&\texttt{select}(Collection,\ Id = value) \Leftarrow \\
&\qquad \textit{compute the page count to be used in yao formula:} \\
&\texttt{CountPage} \quad = \texttt{Collection.TotalSize/PageSize;} \\
&\qquad \textit{compute the oosts:} \\
&\texttt{CountObject} = Collection.CountObject * (value - (Collection.Id.Min) \\
&\qquad /((Collection.Id.Max - (Collection.Id.Min) \\
&\texttt{TotalSize} \quad = CountObject * Collection.ObjectSize \\
&\texttt{TotalTime} \quad = IO * (Collection.TotalSize/CountPage * (1 - exp(-1 * (CountObject/CountPage))) \\
&\qquad +(CountObject * Output)
\end{aligned}
$$

Figure 13: Cost formula for select operator using Yao

time, I/O time, and other overhead is involved in query and result processing. A calibrating procedure is proposed to estimate the coefficients on relational DBMSs, including AllBase, DB2, Informix, and Oracle. In [GST96], an extension of this approach to object systems with experiments on O2 and ObjectStore is reported. This approach has been implemented in the IRO-DB project and has demonstrated its limitations. The main problem is that many data source does not follow the logical cost model formulas, which are not precise enough and derived from more or less extended relational systems behavior. Data sources as files or object databases as O2 or Object Store behave differently from that predicted by the logical cost model. When the number and variety of data sources increase, it becomes difficult to integrate new classes of systems in the mediator. We believe that providing a tool to describe statistics and formulas at the wrapper level, and a consistent way to leverage the mediator cost model with these information is a nice generalization of the calibrating approach.

Another approach is proposed in the HERMES project [ACPS96]. The idea is to record statistics of actual calls to the sources and consequently estimate the cost of the execution plans based on the recorded statistics. More precisely, at the mediator node, a cost vector database is maintained to record information about data source calls as they get executed by the mediator. For each call, the cost vector registers the time to compute the first answer, the time to compute all the answer, the cardinality of the answer, and the type of predicates to which these values correspond to. Summary tables are also generated off-line to avoid heavy burden on

storage. To estimate the cost of a new subquery, the subquery is matched against the cost vector database and a kind of regression is applied. The approach is demonstrated as efficient for sources queried with similar subqueries. We believe this approach very attractive for mediator capable of handling local databases to record cost vectors, but not always possible in case for example of PC-based mediators. Also, if queries differ a lot in qualifications, it is difficult to infer statistics from previous queries. We believe that the so-called caching statistics approach is good to complement a more generic cost model and graciously adapt it to the real source behavior. As mentioned above, specific queries can be recorded at the bottom of our heterogeneous cost model specialization hierarchy, which makes possible to integrate the caching approach for certain data sources.

7 Conclusion

The DISCO project is developing a research prototype of components for searching and integrating information over distributed heterogeneous data sources. The target applications of this project are those of Internet and Intranet which typically require integration of a large number of divers data sources. Since each data source generally performs operations in a unique way, the cost for performing an operation may vary a lot from one wrapper to another. DISCO is addressing this heterogeneous cost model problem through an extensible cost model integrated within the mediator component.

More precisely, we have proposed in this paper a framework for leveraging a generic cost model with more accurate statistics and formulas sent by wrappers at registration time. The framework is general enough to capture and integrate both general cost knowledge declared as rules given by wrapper writers and specific information derived from recorded past queries previously executed. Thus, through an inheritance hierarchy of wrapper descriptions with overriding of cost formulas, which integrates the heterogeneous cost models, the mediator cost computation component can support a wide variety of data sources. We also propose an interface language to export statistics and cost computation rules from a wrapper to the mediator. This language will be semi-compiled in bytecode to be sent efficiently from the wrapper to the mediator at source registration time. It can be seen as useful complements to standard interface description language (e.g., ODMG ODL or CORBA IDL) for giving input to remote query components. In all, the proposed framework is the first to offer a general solution to the heterogeneous cost model problem. This approach is currently being implemented in the DISCO project.

In addition, in this paper, first experiments on top of the ObjectStore database system using the 007 benchmark show a good improvement in performance estimate in comparison to a more classical calibrating approach. We particularly investigate the case of clustering, which can not be easily captured by a calibrating model. Further experiments are currently on their way, particularly on bibliographic and multimedia files, to fully demonstrate the capability of the proposed framework in the case of various sources. In environments with data sources of different functionalities, where each source behave as a specific abstract data type (ADT) on the local collection of objects, the problem of cost evaluation is crucial, for example to avoid processing a large number of images by first selecting a few images from other data source. (See [BRS96, SLR97] for related work in the area of ADTs.) This area is probably where the proposed heterogeneous cost model framework will demonstrate its full power: exporting cost of ADT operations will provide valuable improvement in query optimization. This is a subject of research we intend to address in the near future.

Acknowledgments

The authors wish to thank Philippe Bonnet and Rémy Amouroux for useful comments on previous drafts of this paper.

References

[ABD+92] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The Object-Oriented Database System Manifesto. In *Building an Object-Oriented Database System, The Story of O2*. Morgan Kaufmann, 1992.

[ACPS96] 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*, Montreal, Canada, 1996.

[AHY83] Peter M. G. Apers, Alan R. Hevner, and S. Bing Yao. Optimization Algorithms for Distributed Queries. *IEEE Transactions on Software Engineering*, 9(1), 1983.

[AK93] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In *ACM SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, 1993.

[BGW+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(1), 1981.

[BRS96] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An Open Abstract-Object Storage System. In *ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Canada, 1996.

[C+95] Michael J. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Fifth Int. Workshop on Research Issues in Data Engineering - Distributed Object Management (RIDE-DOM)*, Taipei, Taiwan, 1995.

[Cat95] R. G. G. Cattell. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan-Kaufmann, San Mateo, CA, 1995.

[CDN93] Michael J. Carey, D. DeWitt, and J. Naughton. The 007 benchmark. In *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[Cha82] Jo-Mei Chang. A Heuristic Approach to Distributed Query Processing. In *8th Int. Conf. on Very Large Data Bases (VLDB)*, Mexico City, Mexico, 1982.

[DKS92] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query Optimization in a Heterogeneous DBMS. In *18th Int. Conf. on Very Large Databases (VLDB)*, Vancouver, Canada, 1992.

[DS95] W. Du and M-C. Shan. Query Processing in Pegasus. In *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications.* Prentice Hall, 1995.

[DSH+82] D. Daniels, P. Selinger, L. Haas, B. Lindsay, and P. Wilms C. Mohan, A.Walker. An Introduction to Distributed Query Processing in R*. In *2nd Int. Conf. On Distributed Databases*, Berlin, Germany, 1982.

[G+95] Gardarin et al. IRO-DB: A collection of selected papers. Technical Report RR-95/34, Laboratoire PRiSM-CNRS, University of Versailles, France, 1995.

[GGT95] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. A Cost Model for Clustered Object-Oriented Databases. In *21st Int. Conf. on Very Large Data Bases (VLDB)*, Zurich, Switzerland, 1995.

[GMHI+94] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project - Integration of Heterogeneous Information Sources. In *100th Anniv. Meeting of Information Processing Society of Japan*, Tokyo, 1994.

[GMS94] Cheng Hian Goh, Stuart Madnick, and Michael Siegel. Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment. In *3rd Int. Conf. on Information and Knowledge Management (CIKM)*, Gaithersburg, Maryland, 1994.

[GST96] Georges Gardarin, Fe Sha, and Zhao-Hui Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *22nd Int. Conf. on Very Large Data Bases (VLDB)*, Mumbai (Bombay), India, 1996.

[IP95] Yannis E. Ioannidis and Viswanath Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *ACM SIGMOD Int. Conf. on Management of Data*, San Jose, California, 1995.

[KLSS95] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*, Stanford, California, 1995.

[KTV97] Olga Kapitskaia, Anthony Tomasic, and Patrick Valduriez. Dealing with Discrepancies in Wrapper Functionality. Technical report, INRIA, Rocquencourt, 1997. Submitted for publication.

[LP95] Ling Liu and Calton Pu. Distributed Interoperable Object Model and Its Application to Large-scale Interoperable Database Systems. In *4th Int. Conf. on Information and Knowledge Management (CIKM)*, Baltimore, Maryland, 1995.

[OMG95] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification Revision 2.0.* Object Management Group, Framingham, MA, 1995.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *ACM SIGMOD Int. Conf. on Management of Data*, Montreal, Quebec, Canada, 1996.

[R+96] M. Rusinkiewicz et al. Semantic Integration of Information in Open and Dynamic Environments. Technical report, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1996.

[SAB+97] V. S. Subrahmanian, Sibel Adali, Anne Brink, Ross Emery, James J., Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. HERMES: Heterogeneous Reasoning and Mediator System. submitted for publication, 1997.

[SAD+95] Ming-Chien Shan, Rafi Ahmen, Jim Davis, Weimin Du, and Kent Kent, William. Pegasus: A heterogeneous information management system. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press, 1995.

[SLR97] Praveen Seshadri, Miron Livny, and Ragu Ramakrishnan. The Case for Enhanced Abstract Data Types, 1997. Submitted for publication.

[TRV96] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Heterogeneous Database and the Design of DISCO. In *16th Int. Conf. on Distributed Computing Systems (ICDCS)*, Hong Kong, 1996.

[Wie93] Gio Wiederhold. Intelligent Integration of Information. In *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[Yao77] S. B. Yao. Approximating the Number of Accesses in Database Organizations. *Comm. of the ACM*, 20(4), 1977.

[YC84] C. T. Yu and C. C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16(4), 1984.

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