INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Dealing with Discrepancies in Wrapper Functionality*

Olga Kapitskaia , Anthony Tomasic , Patrick Valduriez

**N° 3138**

Mars 1997

THÈME 1

Rapport de recherche

ISSN 0249-6399

# Dealing with Discrepancies in Wrapper Functionality

Olga Kapitskaia* , Anthony Tomasic* , Patrick Valduriez*

Thème 1 — Réseaux et systèmes
Projet Rodin

Rapport de recherche n° 3138 — Mars 1997 — 21 pages

**Abstract:** Much of the world's information is stored electronically in data sources. The data sources can be full-fledged databases, simple files, HTML pages or specialized data sources that possess diverse query processing capabilities. The common architecture to integrate such sources consists of *mediators* that give a global view over the content of all sources, and *wrappers* that give a local view of each source. Answering queries in this architecture is a difficult problem due to the wide range of capabilities of data sources.

This paper presents a solution to this problem in the context of the Disco query processor. We provide a tool to the wrapper implementor to describe the capabilities of the wrapper in fine detail. When a wrapper is registered with the mediator, the mediator uploads the capabilities of the wrapper, and smoothly integrates these capabilities into query processing. Our solution is novel both in the level of detail permitted by the tool and its easy incorporation into existing query optimization strategies. In this paper we describe: the query processing of Disco, the language for specifying wrapper capabilities, the algorithms that integrates these capabilities into query processing, and an implementation of these techniques in the Disco prototype.

**Key-words:** Heterogeneous Distributed Database, Mediator, Wrapper, Capabilities of Sources, Optimization

*(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

# Gérer les différences des fonctionnalités des adaptateurs

**Résumé :** Aujourd'hui, la plupart des informations se trouve stocké dans diverses sources de données. Ces sources peuvent être des bases de données, des fichiers, des pages HTML ou des banques de données specialisés, chacune possédant des capacités de traitement diffèrentes. L'architecture la plus souvent utilisée pour intégrer de telles sources de données utilise des *médiateurs* qui donnent une vue globale du contenu des sources et des *adaptateurs*, qui donnent la vue locale de chaque source. Dans ce contexte, le traitement de requêtes est rendu difficile par la diversité des capacités des sources de données.

Dans cet article, nous présentons une solution à ce probleme dans le projet DISCO. Nous proposons un outil qui permet aux développeurs d'adaptateurs de décrire les capacités d'une source avec beaucoup de prècision. Lors de l'intégration d'un source, le médiateur intégre la description des capacités, afin de les utiliser lors de l'optimisation des requêtes. Notre solution présente les avantages suivants. D'abord, elle permet une description très fine des capacités des sources. Ensuite, elle permet d'utiliser la connaissance des capacités des sources lors de l'optimisation de requêtes et ceci independemment de la strategie d'optimisation adopté. Ce article decrit (a) le traitement de requetes dans DISCO, (b) le langage de spécification des capacités des adaptateurs, (c) les algorithmes pour integrer ces capacités dans le processeur des requêtes et (d) l'implémentation de ces techniques dans notre prototype.

**Mots-clé :** Base de Données Hétérogène Distribuée, Médiateur, Adaptateur, Capacités des sources, Optimisation

# 1 Introduction

The DISCO (Distributed Information Search COmponents) project [TRV96] is developing components for searching and integrating information over distributed heterogeneous data sources. The data sources can be full-fledged databases, simple files or specialized data servers (e.g. a multimedia server or an information retrieval engine). Thus, data can be structured, semi-structured or unstructured. The target applications of DISCO are those of Internet and Intranet which typically require integration of large numbers of data sources. The main objective of DISCO is to provide uniform and optimized access to the underlying data sources using a common declarative (SQL-like) query language.

To scale up to large numbers of data sources, DISCO adopts a distributed architecture of specialized components [RAH+96, GMHI+94, KLSS95, FRV96] with mediators and wrappers. *Mediators* encapsulate a representation of multiple data sources and provide a query language for the application. They typically resolve conflicts involving the dissimilar representation of knowledge of different data models and database schema, and conflicts due to the mismatch in querying power of each data source. This architecture permits mediators to be developed independently and to be combined, providing a mechanism to deal with the complexity introduced by a large number of data sources. To permit multiple data sources to be accessed in a uniform way, mediators accept queries and transform them into subqueries that are distributed to data sources.

To deal with the heterogeneous nature of data sources, *wrappers* give a structured view of the data source and transform subqueries from a subset of the mediator query language to the particular language of the data source. A wrapper supports the functionality of transforming queries appropriate to the particular data source, and reformatting answers (data) appropriate to each mediator. The wrapper implementor (DBI) writes wrappers for each type of data source.

The wrapper interface is a key to both query processing efficiency and wrapper development effort. There are two extreme solutions for the wrapper interface: high-level and low-level. A high-level (SQL-like) interface, e.g., Open Database Connectivity (ODBC), is well-suited for modern databases, e.g., relational databases, since the wrapper interface can be almost directly mapped in the database interface. However, a data source may have a very limited interface, for instance, queries of the form "select * where att matches value" on an information retrieval (IR) engine. In that case, writing the wrapper involves a lot of effort to support complex SQL statements using the data source interface. For instance, mapping project or multi-attribute select in the IR engine interface is quite involved, even using a *wrapper implementation toolkit* [PGGMU95]. A low-level interface, e.g., scan, obviously makes wrapper development easy but makes query processing inefficient by forcing the data sources to be copied in the mediator for further processing.

The problem with either high-level or low-level wrapper interface is that the interface is fixed. To help the DBI, DISCO provides a flexible wrapper interface. DISCO interfaces to wrappers at the level of an abstract algebraic machine of logical operators [Gra93]. When the DBI implements a new wrapper, she chooses a (sub) set of logical operators to support. The DBI implements the logical operators, and also implements a call in the wrapper interface which returns the set of supported logical operators.

Such operator-based interface solves the classical problem of varying level of functionalities in data sources. It provides a good balance between implementation of new complex interfaces and the gain from additional complexity. It also permits incremental wrapper implementation (going from simple operators to more complex ones). However, it introduces a new problem for mediators which must produce code for wrappers of varying functionality. We address this problem by having the mediator query processor aware of each wrapper functionality (in terms of supported operators) and use this knowledge in processing queries in a way which fully exploits the wrappers. For example, a mediator may generate a logical expression for a wrapper to project the `name` attribute from a relation `r`.

```
project(name, scan(r))
```

The mediator will pass this logical expression to a wrapper, thereby, pushing the project operation onto the wrapper, only if the wrapper interface supports the `project` and `scan` logical operators, and only if the wrapper supports composition of these logical operators.

To summarize, the contributions of this paper are the following. First, we define an operator-based model as wrapper interface. This interface is the basis for a flexible language to specify the functionality of wrappers. Second, we propose algorithms for combining wrapper functionality with mediator query processing. We believe they work with any query processing algorithm that can accommodate logical operators. Finally, we describe the current implementation of the operator model and query processing algorithm in the DISCO prototype. All examples used to illustrate the algorithms are based on our prototype implementation.

This paper is organized as follows. Section 2 describes mediator query processing. Section 3 states the problem formally. Section 4 proposes algorithms for combining wrapper functionality with mediator query processing. Section 5 describes our language for specifying the wrapper interface and treats the question of logical operators composition. Section 6 describes the current implementation status of the operator model in the Disco prototype. Section 7 discusses related work and Section 8 concludes the paper.

# 2 Mediator Query Processing

The mediator includes a query processor and a run-time system. It also contains an internal database which records information on data sources, types, schemas, and views. This database is updated during *registration* phases when new wrappers for data sources are introduced by importing their local schema and functionality. To hide local schema dissimilarities to the application, views are used. The mediator implements physical operators for all supported logical operators. The run-time system is able to execute all physical operators that implement the logical operators of the wrapper interface. The query processor searches for the best way to balance the execution of an input query between the wrappers and the run-time system. In this section, we provide an overview of query processing in Disco and describe the way wrapper interaction is modeled.

## 2.1 Overview

Query processing consists of generating the best execution plan for a query and executing that plan using the wrappers and the mediator run-time system. The input query explicitly references the views defined over the local schemas. Query processing proceeds along several steps (see Figure 1).

Step 1 corresponds to semantic query processing in traditional optimizers. It parses the query and, using the view definitions, reformulates it into one or more equivalent queries on the local schemas. This is done by using view definitions [TRV96, FRV96]. Each query is decomposed into $n$ sub-queries each expressed on a local schema (for a wrapper) and a *composition query*. The composition query is to be executed by the mediator after receiving the results of execution of all sub-queries by the wrappers. Since multiple composition queries are possible, depending on the grouping done for the sub-queries, we use the heuristic of [LRO96] to consider only the queries that issue as few sub-queries as possible to the same data source. At this stage, query processing ignores the wrappers functionality. Thus, the sub-queries assume that all logical operators are supported by each wrapper.

Step 2 performs logical search space generation. It transforms a decomposed query (i.e., the sub-queries and the composition query) into a logical operator tree. Disco has the usual logical operators of `project`, `join`, etc. *Transformation rules*, such as commuting and associating join, rewrite logical expressions to equivalent logical expressions. For each wrapper, there is a sub-tree of the logical operator tree that corresponds to a *preliminary* logical plan for that wrapper. Another (top-level) sub-tree corresponds to the *preliminary* composition query plan. These plans are preliminary because we don't know yet whether a wrapper is capable of executing its corresponding plan.

Step 3 is new and is addressed in this paper. Based on the knowledge of each wrapper functionality, it identifies the parts of a preliminary logical plan that can be actually executed by the corresponding wrapper. Each part of the logical operator tree that can be executed by a wrapper is kept as a candidate sub-query, and the rest of the tree that cannot becomes part of the composition query plan. At the end of this step, a valid logical operator tree for the original query has been generated.

Step 4 corresponds to the traditional step of execution plan generation. The distributed execution plan (corresponding to the sub-queries and the composition query) is generated by transforming logical operators into physical operators such as index-scan, hash-join, etc. Since each logical operator may have several physical implementations, many physical operator trees can be produced. Disco uses a cost model [NGT97] to select the physical operator tree with least cost.

Step 5 corresponds to distributed query execution. The query processor calls the wrappers to execute their corresponding execution plan and to send the results back to the mediator run-time system which can compute the final composition query.

## 2.2 Wrapper Call

To model a call to a wrapper we introduce the logical operator `submit(id, expr)` that takes a source id and a logical operator expression as arguments. This operator means that the given logical expression `expr` will be

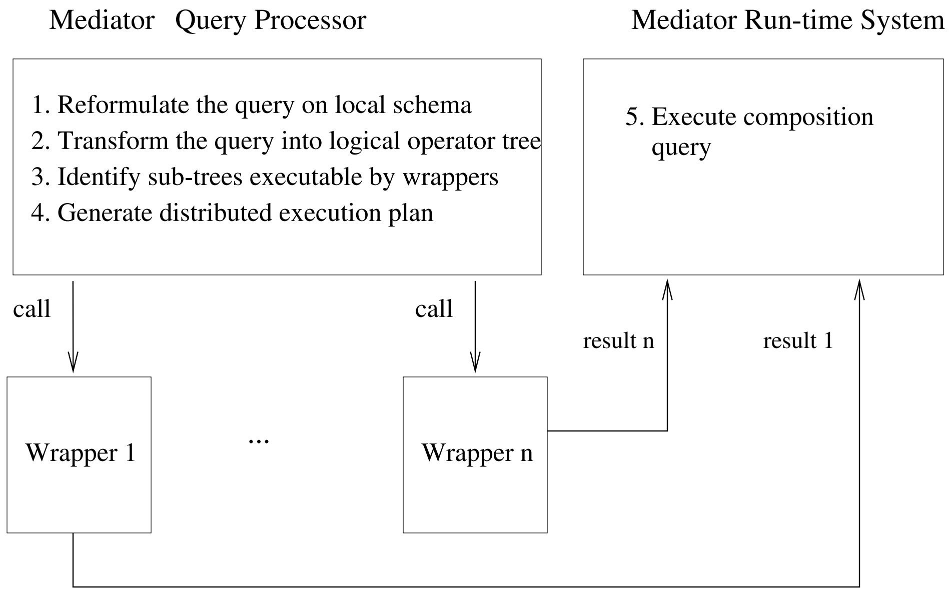


Figure 1: Query Processing in DISCO

executed on a source `id`. Each access to a local data source (wrapper) is represented by `submit`. For example, the logical expression

```
project(map, join(pred, submit(source1, select(scan(A))), submit(source1, scan(B))))
```

means that two logical expressions will be evaluated at `source1`: `select(scan(A))` and `scan(B)`. The results of these expressions, say `r1` and `r2`, are sent to the mediator. Having received the results, the mediator executes the rest of the expression: `project(map, join(pred, r1, r2))`.

Technically, the addition of the `submit` operator to a logical operator tree $t$ at subtree rooted at an operator $o$ transforms the tree into a new tree $t'$. The new tree $t'$ is defined as $t$ with subtree rooted at $o$ replaced by the `submit` operator. Thus, the `submit` operator is a leaf node of the new tree. The subtree rooted at $o$ is passed as *argument* to the `submit` operator. Thus, no information is lost since the subtree is preserved. For the rest of this paper, by an abuse of notation, we consider the new tree $t'$ to be $t$ with an additional `submit` operator inserted into the tree, and we ignore the shift of the subtree to the argument of the operator.

# 3 Problem Definition

This paper concentrates on Step 3 of query processing (Figure 1), the generation of the preliminary logical plans that are executed by the corresponding wrappers. A preliminary logical plan for a wrapper is represented by a logical operator tree. During the registration phase the mediator imports a wrapper's functionality in the form of a grammar. The grammar describes in a compact way the logical operators supported by the wrapper, the specific arguments permitted to each operator, and the relationships permitted *between* operators. Thus, the wrapper defines the expressions of logical operator trees it can process. The set of all expressions that the wrapper can process form the language conforming to the imported grammar.

Currently in DISCO, the set of logical operators exported by a wrapper must be a sub-set of the logical operators supported by the mediator. The problem of this paper is, given a logical operator tree generated during logical search space generation that *should* be executed by a wrapper, find a sub-tree that the wrapper *can* execute. Finding an executable sub-tree amounts to introducing an operator `submit` in the right place in a logical operator tree. This section defines the problem formally and identifies the solution we provide.

Let us introduce the following definitions:

$UA_{med}$ - set of mediator logical operators, (i.e., the operators supported by the mediator)

$UA_{wrapp_i}$ - set of $wrapper_i$ logical operators (i.e., the logical operators supported by $wrapper_i$)

$L(UA_{med})$ - mediator logical operators language (i.e., the set of all possible logical expressions (operator trees) that can be generated by the mediator in Step 2)

$L(UA_{wrapp_i})$ - $wrapper_i$ logical operators language (i.e., the set of all logical expressions that $wrapper_i$ can execute)

$L(Q)$ - the user query language (i.e., the set of all queries expressed in a high-level language)

We model Steps 1 and 2 of query processing as a rewriting function (see Figure 1). This function rewrites a user query $q$ into a set of equivalent logical operator trees. Each logical operator tree consists of preliminary logical operator trees for the data sources and a preliminary composition query.

*Rewrite* : L(Q)$\rightarrow 2^{L(UA_{med})}$, where $2^{L(UA_{med})}$ denotes the set of all subsets of $L(UA_{med})$.

Thus, the rewrite function takes as input $q \in L(Q)$ and returns a set of $l \in L(UA_{med})$, where $l \in$ *Rewrite*$(q)$. In addition, the preliminary logical operator tree $l_{wrapper_i}$ for each wrapper$_i$ has been identified.

The set of logical operators exported by wrapper$_i$ is a subset of the set of logical operators supported by the mediator. Thus, $wrapper_i$ can execute only a proper subset of the mediator expressions:

$(UA_{wrapp_i} \subset UA_{med}) \Rightarrow (L(UA_{wrapp_i}) \subset L(UA_{med})) \Rightarrow (\exists l \mid l \in L(UA_{med})$ and $l \notin L(UA_{wrapp_i}))$

The problem is to find a rewriting $l'_{wrapper_i}$ of each preliminary logical operator tree $l_{wrapper_i}$ such that all but the leaf nodes of $l'_{wrapper_i}$ is an expression in the mediator logical operator language, and the leaf nodes are operators `submit` whose argument is an expression in a wrapper logical operator language. The part of $l_{wrapper_i}$ that cannot be executed by the wrapper, say $l''$, is executed by the mediator by extending the preliminary composition query with $l''$.

To resolve this problem we define a transformation $T$ that performs this rewriting for all $l_{wrapper_i}$. (The algorithms implementing this transformation are given in the next section.) Transformation $T$ satisfies the following definition:

$T : L(UA_{med}) \rightarrow L(UA_{med} \cup \texttt{submit})$

In addition, since we are in an environment where mediators issue subqueries to wrappers and wrappers return subanswers, the transformation $T$ maintains the following two invariants : (i) each traversal path from the root operator to a leaf operator in the generated tree contains exactly one `submit`, and (ii) the subtree argument to `submit` contains only operators permitted by the source. The first invariant means that a submit operator cannot contain another submit operator, nor can some leaf operator have a `submit` missing. The second invariant means that the source will only receive subqueries that it is capable of executing.

# 4 Producing Plans Executable by Wrappers

In this section, we describe two algorithms for producing final logical plans, i.e., logical operator trees, that can be executed by wrappers (for step 3 in Figure 1). Algorithm `allTrees` generates all possible trees while algorithm `maxTree` generates only one tree.

## 4.1 Assumptions and Definitions

For the $UA_{wrapp_i}$, we define a very simple interface, a set of operator names. Each operator has a specific interface that encodes its meaning, signature and collection of possible arguments. For now, we assume that if the wrapper exports the operator name, it means that it supports the *entire* definition for that operator. For example, if a source exports the `select` operator name in its $UA_{wrapp_i}$, it must support the entire definition of `select`. We will relax this restriction later, in Section 5. For Disco, `select` supports boolean conjuncts and disjuncts in the predicate. Thus, there is a *universal* definition of `select` known to the mediator and all wrappers that support that operator. We also make an assumption that each source exports at least the `scan` operator in the $UA_{wrapp_i}$.

We consider two possible strategies of adding the `submit` operator to a preliminary logical operator tree. One produces all possible trees that can be executed on a source, i.e., generate all possible combinations of logical operator trees that differ by the place of the `submit` operator in the tree. The other produces the tree with the maximum number of operators executed on the wrapper. Both strategies are useful. The first strategy explores all possible plans, including plans where an operator is executed on the mediator even though the wrapper may execute it. This plan may be interesting if the cost of executing of the operator in the mediator is less than that in the wrapper. However, exploring these plans may well increase optimization time. The second strategy implements a heuristic that reduces network traffic.

```
(Set, boolean) allTrees(Wrapper w, LogicalOperator tree, Set capabilities) {
    op = tree.topOperator();
    if (op.children == 0) // must be a scan and must be in capabilities
        return ( (new Set(new Submit(w, op))), false);
    else
        b = false;
        S = ∅;
        foreach (i∈ op.children())
            ( Set_i, hasSubmit ) = allTrees(w, i, capabilities);
            S = S ∪ ( Set_i, hasSubmit );
            b = b ∪ hasSubmit;
        ResultSet = crossProduct( Set_1, ...  ,Set_n, op );
        if (b)
            return (ResultSet, true);
        else if (op ∈ capabilities)
            ResultSet = ResultSet ∪ new Submit(w, op);
            return (ResultSet, false);
        else // op is not in capabilities
            return (ResultSet, true);
}
```

Figure 2: Algorithm for exhaustive tree generation

The function `allTrees` implements the first strategy and has the following signature:
`allTrees` : $L(UA_{med}) \rightarrow 2^{(L(UA_{med} \cup submit))}$. We believe that the algorithm is both sound and complete with respect to the transformation $T$, that is $(l' \in$ `allTrees` $(l)) \Leftrightarrow (l \xrightarrow{T} l')$.

The function `maxTree` implements the second strategy and has the following definition:
`maxTree` : $L(UA_{med}) \rightarrow L(UA_{med} \cup submit)$.

We believe that the algorithm is sound with respect to the transformation $T$, that is
$(l' =$ `maxTree`$(l)) \Rightarrow (l \xrightarrow{T} l')$

## 4.2 Exhaustive tree generation

### 4.2.1 Algorithm `allTrees`

The algorithm for producing all possible executable trees performs one pass. All trees are being constructed dynamically at the same time. The algorithm proceeds bottom-up and consists of two sequences, the initial *check* sequence followed by a *copy* sequence (see Figure 2). Each sequence is performed in a number of steps. Each step of the check sequence corresponds to examining an operator to see if it can be executed by the source or not. Each step of the copy sequence corresponds to copying the current operator into the output tree. The algorithm operates in the check sequence if all operators considered so far can be executed by a source. The switch to copy sequence occurs when an operator is found in the tree that cannot be executed by the source. In this case, all remaining operators must be executed in the mediator.

The input to the algorithm is the preliminary logical operator tree $l_{wrapper_i}$ for a wrapper$_i$ and the set of capabilities of the source UA$_{wrapper_i}$. All operators apply only to the data on the associated source. The output of the algorithm is a pair, consisting of a set of new logical operator trees (containing `submit`) and a boolean indicating if the original logical operator tree contains at least one operator that cannot be executed on the source. Note, that since each source exports at least `scan` operator, it is always possible to produce at least one new logical operator tree. The development of the algorithm is illustrated by the examples in Figures 3 and 4.

The algorithm starts operating in the check sequence. It proceeds by traversing the input tree in post-fix order. After visiting each child, the algorithm tests the result of the recursive call on each child to see if an operator that cannot be executed on the source has been detected in that child. If no operator has been detected,

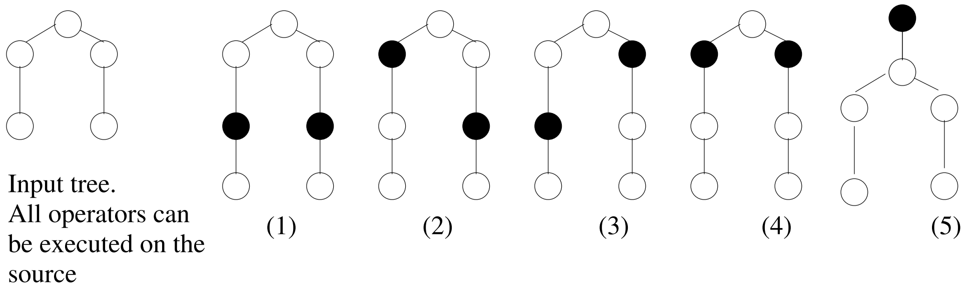


Figure 3: Logical operator trees (1)-(5) are the result of applying the `allTrees` function to the input tree. All logical operators in the input tree can be executed on the source. White circles represent operators, black circles represent the `submit` operator.

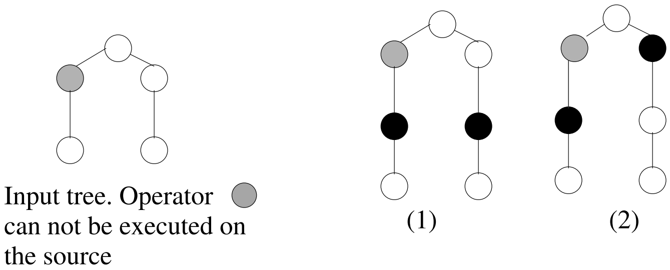


Figure 4: Logical operator trees (1)-(2) are the result of applying the `allTrees` function to the input tree. Marked logical operator cannot be executed by the source.

i.e., all operators so far can be executed by the source, one of two cases arises: either the source is capable of executing the current operator or it is not. If it is, the algorithm remains in the check sequence. It returns the set of new logical operator trees resulting from cross-product of the sets of new logical operator trees produced by recursive calls on each child, and a new logical operator tree produced by adding `submit` to the subtree of the input logical operator tree rooted at the current operator. The boolean value returned is *false*, indicating that all operators can be executed.

If the source is not capable of executing the current operator, the algorithm switches to the copy sequence. It returns the set of new logical operator trees resulting from cross-product of the sets of new logical operator trees produced by recursive calls on each child. In addition, the boolean value returned is *true*, indicating that an operator has been found that cannot be executed by the source.

If the result of the recursive call on the children detects an operator that cannot be executed by the source, the algorithm switches to the copy sequence. In this case, the result is the set of new logical operator trees resulting from cross-product of the sets of new logical operator trees produced by a recursive call on each child, and the boolean value is *true*. Thus, the switch from the check sequence to the copy sequence means that, at each step of recursion, the algorithm cannot add the operator `submit` to the input tree at the subtree rooted at the current operator.

Function `crossProduct` used in the algorithm of Figure 2 produces the cross product of the given sets. The input to the function is a set of sets of logical operator trees $\{Set_1, ..., Set_n\}$, and a logical operator *op*. The output is a set of new logical operator trees $Set = \{op_1, ..., op_m\}$, each having an operator *op* as a root, and the children of each new logical operator tree are one of possible combinations of cross product. The number of new logical operator trees produced by crossProduct is $m = \prod_{k=1}^{n} card(Set_k)$, where $card(Set_k)$ is the number of logical operator trees in $Set_k$ and $n$ is number of sets.

### 4.2.2 Examples

We illustrate the algorithm `allTrees` with the following examples. [1] The logical operators supported at the mediator level are:

$UA_{med} = \{$`select, project, join, scan`$\}$.

[1]All examples given in this section are the output of our prototype.

**Example 1.** The input query $q$ is:

```
select x.name
from   x in books
where  (x.name="franklin")and(x.year<1995)
```

For simplicity, we suppose that query $q$ concerns one wrapper $w$. Let us consider the preliminary logical plan $l$ = `project(select(scan(books)))` and two cases of wrapper functionality.

1. $UA_{wrapper} = \{$`select, project, join, scan`$\}$.

The algorithm first introduces the `submit` operator at the sub-tree rooted at `scan`. Since the parent of the `scan` operator, `select`, is unary, only one $Set(i)$, containing a logical expression $l'$ = `submit(w, scan)`, is produced. Function `crossProduct` gets as input a set containing this $Set$ and a logical operator `select`. The result of `crossProduct` is a $Set'$ containing one logical expression: $l''$ = `select(submit(w,scan(books)))`. Since the algorithm stays in the check sequence, one more logical expression, resulting from introducing the operator `submit` at a sub-tree rooted at `select` is produced : $l'''$ = `submit(w, select(scan(books)))`. Thus, the `ResultSet` on this step contains two logical expressions :

```
submit(w,select(scan(books)))
select(submit(w,scan(books)))
```

The algorithm continues the execution in the same way. The final result is the following $Set$ of logical operator trees:

```
project(submit(w,select(scan(books))))
project(select(submit(w,scan(books))))
submit(w,project(select(scan(books))))
```

2. $UA_{wrapper} = \{$`project, scan`$\}$.

The result of applying the algorithm is the $Set$ that contains only one logical operator tree:

```
project(select(submit(w,scan(books))))
```

**Example 2.** The input query $q$ is:

```
select x.title
from   x in books and y in books
where  x.name=y.name
```

As in the previous example, we suppose that the query $q$ concerns only one wrapper $w$. Let us consider the preliminary logical operator tree: $l$ = `project(join(scan(books)scan(books)))` and two cases of wrapper functionality.

1. $UA_{wrapper} = \{$`project, select, join, scan`$\}$.

The result of applying the algorithm is the following $Set$:

```
submit(w,project(join(scan(books)scan(books))))
project(submit(w,join(scan(books)scan(books))))
project(join(submit(w,scan(books)),submit(w,scan(books))))
```

2. $UA_{wrapper} = \{$`join, select, scan`$\}$.

Then the result of applying the algorithm is the following $Set$ :

```
project(submit(w,join(scan(books)scan(books))))
project(join(submit(w,scan(books))submit(w,scan(books))))
```

## 4.3 Single tree generation

### 4.3.1 Algorithm maxTree

Algorithm `maxTree` pushes as many operators as possible into the `submit` operator in order to increase local processing in the wrappers and reduce network traffic. The algorithm performs one pass. It proceeds bottom up and operates in two sequences, the initial *check* sequence followed by *copy* sequence (see Figure 5). The meaning of these sequences is the same as for algorithm `allTrees`.

The input to the algorithm is the logical operator tree and the set of capabilities of the source. All operators apply only to the data of the associated source. The output of the algorithm is a pair consisting of the new logical operator tree and a boolean indicating if the new logical operator tree contains a `submit` operator somewhere in it. The algorithm starts with the check sequence. It proceeds by traversing the tree in post-fix order. After visiting each child, the algorithm tests the result of the recursive call on each child to see if a `submit` operator has been added anywhere. If so, the algorithm switches to the copy sequence, the `submit` operator is added to all children that do not have one, and the current operator is returned with all the modified children as its argument. In addition, the boolean value returned is *true*. If no child has a `submit` operator, then one of two cases arises: either the source is capable of executing the current operator or it is not. If it is, the algorithm remains in check sequence and the operator and its children are returned with a *false* boolean value. If it is not, a `submit` operator is added to all children, the algorithm switches to the copy sequence, and the current operator is returned will all modified children as its argument and the boolean value is *true*.

```
LogicalOperator maxTree(Wrapper w, LogicalOperator tree, Set capabilities) {
    (newtree, hasSubmit) = addSubmit(w, tree, capabilities);
    if (hasSubmit)
        return newtree;
    else
    return new Submit(w, newtree);
}

(LogicalOperator, boolean) addSubmit(Wrapper w, LogicalOperator tree, Set capabilities) {
    op = tree.topOperator();
    if (op.children == 0) // must be a scan and must be in capabilities
        return (tree, false);
    else
        b = false;
        S = ∅;
        foreach (i ∈ op.children())
            (Set_i, hasSubmit) = addSubmit(w, i, capabilities);
            S = S ∪ (Set_i, hasSubmit)
            b = b ∪ hasSubmit;
        if (!b)
            if (op ∈ capabilities)
                return (tree, false)
            else
                return (new Submit(w, tree), true)
        else
            foreach (i ∈ S)
                if i's flag is false, add Submit to i's operator
            return (new Operator(op, S), true)
}
```

Figure 5: Algorithm for generating a single tree with the maximum work done by the wrapper.

### 4.3.2 Examples

Let us consider the following examples. The logical operators supported at the mediator level are:

$UA_{med}$ = { select, project, join, scan }.

**Example 1.** The input query $q$ is:

```
select x.name
from   x in books
where  (x.name="franklin")and(x.year<1995)
```

For simplicity, we suppose that query $q$ concerns one wrapper $w$. Let us consider the logical operator tree $l$ = `project(select(scan(books)))` and two cases of wrapper functionality.

1. $UA_{wrapper}$ = {select, project, join, scan }.

Then the result of applying the algorithm `maxTree` is the following:

```
submit(w,project(select(scan(books))))
```

2. $UA_{wrapper}$ = {project, scan}.

Then the result of `maxTree` is:

```
project(select(submit(w,scan(books))))
```

**Example 2.** The input query $q$ is :

```
select x.title
from   x in books and y in books
where  x.name=y.name
```

As in the previous examples, we suppose that the query $q$ concerns only one wrapper $w$. Let us consider the logical operator tree $l$ = `project(join(scan(books)scan(books)))` and two cases of wrapper functionality.

1. $UA_{wrapper}$ = {project, select, join, scan}.

Then the result of `maxTree` is:

```
submit(w,project(join(scan(books)scan(books))))
```

2.$UA_{wrapper}$ = {join, select, scan}.

Then the result of `maxTree` is:

```
project(submit(w,join(scan(books)scan(books))))}
```

### 4.4 Discussion

Essentially, algorithms `allTrees` and `maxTree` are the functions used by an optimizer to translate a preliminary logical operator tree $l$ for a wrapper into another logical operator tree $l'$ that can be executed. The resulting logical operator trees contain two parts, the part $a$ executed by the mediator and the part $b$ executed by the wrapper. Since $l$ is simply a subtree of the overall tree used for executing the entire query, by simply replacing $l$ by $l'$, the composition query for the overall query is changed – it is extended by $b$ on the branch that contained $l$.

Once this step is accomplished for all preliminary logical operator trees, the resulting tree can be executed. This logical operator tree then goes through the physical optimization step and the resulting (again, one or more) physical operator trees can be assigned a cost. The optimizer repeats this procedure in some way to search for the lowest cost physical operator tree. The way used to search depends on the optimizer: randomized, dynamic programming, etc. All of these optimizers can benefit from the algorithms we provide.

## 5 Wrapper Interface Language

In Section 4, we defined a very simple interface to express wrapper functionality: a set of operator names. If a wrapper exports an operator it means that it supports the entire interface of this operator.

This assumption is too restrictive for the real data sources. Very often wrappers are able to support a given operator but only on a *subset* of the interface of the operator. The examples of restrictions that have to be taken into account are the supported path expressions, boolean conjuncts, boolean disjuncts, the set of predicates that can be applied on the attributes, variable bindings for a particular attribute, etc.

In this section, we extend the wrapper interface with more functionality. We still use an operator-level interface; a wrapper provides the description for each logical operator it supports. A wrapper exports its *functionality description* during the registration phase of query processing and this description is used in query processing to generate logical operator trees executable on the source. We first extend the description to deal with the particular nature of each operator, and then we extend the description to deal with the composition of operators.

## 5.1 Operator Description Language

The wrapper functionality description expresses several important facts about the underlying sources. First of all, if a wrapper exports several collections, a given logical operator might be supported by one collection but not by the other. A wrapper can specify which collections support which logical operators. Another very important restriction for heterogeneous sources are on predicates: some operators require certain comparisons to appear in predicates. Consider wrappers for information retrieval (IR) systems. For example, a WAIS engine takes as input a word (or sometimes a list of words with boolean connectives). This means that a wrapper for the WAIS engine requires that a `select` logical operator is always sent to a wrapper. Since the `select` operator contains a predicate and the predicate compares an attribute to a constant, the wrapper for the the WAIS engine will always be able to convert the subquery into a call to the WAIS engine.

The wrapper can also require that only a limited set of predicates be applied on each given attribute for a given logical operator. For example, a WAIS engine understands only equality predicate for any attribute in the query, and does not understand greater than or less than comparison predicates.

To understand the flavor of the wrapper interface, let us consider the following simple example. The Database and Logic Programming (DBLP) bibliographic server[2] is an IR system that processes two IR requests.[3] Given the name of an author, it returns the list of publications of this author; given a keyword it returns the list of publications with the keyword in the title of the publication.

A wrapper for DBLP exports the following simple schema (in ODL-like language)

```
interface Publication (extent publications) {
   attribute string Title;
   attribute string KeywordTitle;
   attribute string Author;
};
```

The wrapper for this data source exports the following logical operators: {`select`, `project`, `scan`}. The `select` operator can be applied on the collection `Publication`. More precisely it is applied on the attributes `KeywordTitle` and `Author`. Since a DBLP server accepts only one constant as input, the wrapper accepts a predicate on the `select` with exactly one of the attributes `KeywordTitle` or `Author`. The DBLP server uses information-retrieval methods of search, thus retrieving only *matching* publications. Here, the wrapper implementor has two choices. One choice is to filter the results of the IR search, checking for equality. Thus, exactly the meaning of the equality predicate is implemented. A second choice is to pass every answer from the IR search back to the mediator. In this case, the wrapper essentially exports a *match* predicate instead of equality. Since the DBLP IR engine always returns the entire publication, the `project` operator is implemented in the wrapper. In addition, we require that the project operator is always over attributes `Author` and `Title`. In effect, the attribute `KeywordTitle` does not exist in the underlying data source. We use this attribute to model the fact that IR searches match any keyword in a title. Thus, the user does not give the entire title of a publication in a query, only a keyword can be specified. The operator `scan` can be applied to the extent `publications` without further restrictions.

In our wrapper interface language, this functionality is expressed as follows:

```
select     [publications 1 { bind Author (=)
                             bind KeywordTitle (=)
                           }]
project    [publications 2 { bind combine Author ()
                             bind combine Title ()
                           }]
scan       [ ALL ]
```

[2]The DBLP bibliographic server is located at `http://sunsite.informatik.rwth-aachen.de/dblp/db/index.html`.

[3]We consider only part of the server's functionality.

Each operator expresses its functionality separately. For each operator, the operator lists the collection names and attributes that it can understand. Thus, for a given operator tree, each occurrence of an attribute is traced to the original collection that the attribute is derived from. This pair of (attribute, collection) is compared against the functionality for the operator. If the operator lists ALL, as in the scan operator, then all collections are permitted.

The integer `1` after the collection name is the minimal number of the attributes of this collection to appear in the argument of the operator. The keyword bind before the attribute name means that this attribute must appear in the attribute of the operator. The absence of the keyword combine means that the attribute cannot be combined, i.e., only one attribute in a query can be bound. For each attribute, the list of supported predicates for any given logical operator is given after the attribute name. ( "=" for all attributes of DBLP). The keyword ALL in the description of logical operator `scan` means that this operator can be applied to any attribute with any predicates. In general, the keyword ALL can replace (i) the list of all collections on which operator can be executed, or (ii) the list of attribute names, or (iii) the list of the predicates supported. The complete BNF for the operator description language is given in Appendix A. The functionality of this language is inspired by [LRO96].

This operator description language has several advantages. First of all, it facilitates the task of integrating a new wrapper. Expressing details at a high-level about the wrapper functionality is easy. Second, the level of detail is high, thus giving a finer control of the implementation of a wrapper. Third, the wrapper interface is dynamic. It is uploaded during the registration phase and does not have to be compiled. If the functionality of a wrapper changes, a new functionality description is uploaded dynamically and replaces the old one. Finally, the wrapper interface is general. It provides a uniform description of any set of logical operators supported by a wrapper (relational logical operators as well as non-relational ones).

## 5.2 Integration into Query Processing

The wrapper interface language allows a wrapper to define in detail its functionality. This information is used in the algorithms of Section 4. Modification of the algorithms to account for the extended functionality of this section is straightforward. When the algorithms check that an operator can be executed by a source, the algorithm also checks that the arguments of the operator match the restrictions of the new functionality.

That is, the initial check sequence of the algorithm verifies if the current logical operator satisfies the wrapper functionality description, i.e., if the collection name mentioned in the query is in the list of collection names to which the current operator can be applied, if the attribute names are in the list of the attribute names to which the current operator can be applied, and if the predicate in the query is supported for the given attribute. In addition we verify if the binding requirements for the logical operator hold, i.e., if a necessary number of attributes appear in the operator argument and if these attributes are combined in a required way.

In the current implementation, we treat the boolean conjuncts and disjuncts in a special way. The functionality language in its current stage does not provide the way to specify if a wrapper supports conjuncts and disjuncts. Thus, for an expression `select(''x.age > 25 and x.salary=50000'',` *op* `)`, for some operator tree $t$, a check is done separately for both `x.age > 25` and `x.salary=50000`. The operator `select` can be pushed inside `submit` if both conjuncts of the predicate satisfy the exported wrapper functionality. In future work, we are planning to extend the language to describe restrictions on conjuncts and disjuncts.

## 5.3 Operator Composition

Query processing and the algorithms of Section 4 both assume that all wrappers support the `scan` operator. All wrappers must export this operator in their functionality description. While all wrappers support `scan`, not all of them can execute a subquery consisting only of a `scan`, since the result of the `scan` operator is the entire collection being accessed. Some wrappers are not able to provide the entire data collection of underlying sources. For example, a DBLP wrapper can only execute queries that contain some selection criteria (one of the `KeywordTitle` or `Author` should appear); it is not able to return the entire collection of the underlying publications as an answer. Other wrappers are likely to have numerous other subtleties in the form of logical operator trees they can execute.

To express the requirement that `select` must appear with `scan`, we define a language for expressing restrictions on *operator composition*. The wrapper implementor specifies the grammar that describes all possible forms of logical operator trees that a wrapper can understand. She specifies the restrictions in operator composition, the possibilities for a logical operator to be an argument of another logical operator, etc. For example, the

DBLP wrapper exports the grammar[4] shown in Figure 6. This grammar states that `scan` must be composed with `select`. The parser generated from this grammar parses over a string that lists each operator with its arguments, e.g. `select(scan)`.

```
TOKEN :
{} {
      <OP1 : "select" >
   |  <OP2 : "scan" >
   }

void Operators() :
{}
{
  <OP1> "(" <OP2> ")"  <EOF> {}
}
```

Figure 6: Composition grammar for DBLP wrapper

A wrapper can choose not to specify its requirements of operator composition order. If no specification is provided, a default grammar is used to check the candidate logical plans. In the current implementation we use “relational-style” operator composition rules: any logical operator (except `scan`, that should have an extent name as argument) can have any other logical operator as argument and there are no specific restrictions on their composition.

The advantages of the composition functionality grammar are similar to the advantages of the operator functionality grammar: better control over the wrapper implementation, dynamic modification of the functionality, etc.

### 5.4 Extending Algorithms `allTrees` and `maxTree`

Extending algorithm `allTrees` to understand the composition functionality is straightforward. The algorithm is extended to verify each generated logical operator tree. For each logical operator tree, the argument of the `submit` operator is converted to a string and parsed by the corresponding parser. If this string is parsed successfully, the logical operator tree, argument of operator `submit`, satisfies the composition order requirements of the wrapper. Thus, the generated logical operator tree is kept in the result set. If the string does not parse, the argument of `submit` does not satisfy the composition requirements of the wrapper, and the generated logical operator tree is removed from the result set.

Extending algorithm `maxTree` is more involved and requires a new algorithm. The new algorithm (that we call `maxTreeExtended` ) performs two passes, an initial bottom-up traversal of the tree followed by a top-down traversal. The bottom-up traversal operates like `maxTree` algorithm, except that it does not construct the new tree during this pass. Each node is simply marked with a boolean variable *mark* indicating if the node was in the *check* or *copy* state of the bottom-up traversal. (Let us remind that the node is in the *check* state if all operators so far can be executed by the source, and the node is in the *copy* state if an operator was found that cannot be executed by the source.) The top-down traversal descends the tree, examining each node's state as indicated by the boolean variable in the first traversal.

If the node is in a *copy* state the top-down traversal recurses on the node's children and then copies the results of the recursive calls into the result tree, just as the algorithm `maxTree`. If the node is in the *check* state the algorithm constructs a string representing the subtree rooted at the current node, and attempts to parse it. If the parsing is successful, a `submit` operator is added to this node and the algorithm returns this operator. If the parsing is unsuccessful, the current node is copied into the result tree and the top-down traversal continues on each child branch independently.

The result of the bottom-up and top-down traversals of the algorithm insures that at most a single `submit` lies on each path from the root to each leaf. However, depending on the tree and the operator composition specified by a wrapper, some branches might not have a `submit` operator. Thus, the algorithm has to check the result tree to insure that each path has a `submit`. If a path without `submit` operator exists, the tree cannot be labeled properly with `submit` operators, and `maxTreeExtended` returns NULL. The resulting algorithm is given in Appendix B. The development of the algorithm is illustrated by the example in Figure 7.

[4] The grammars are given in the syntax of Jack Parser generator (http://www.suntest.com/Jack/) that we use in our prototype

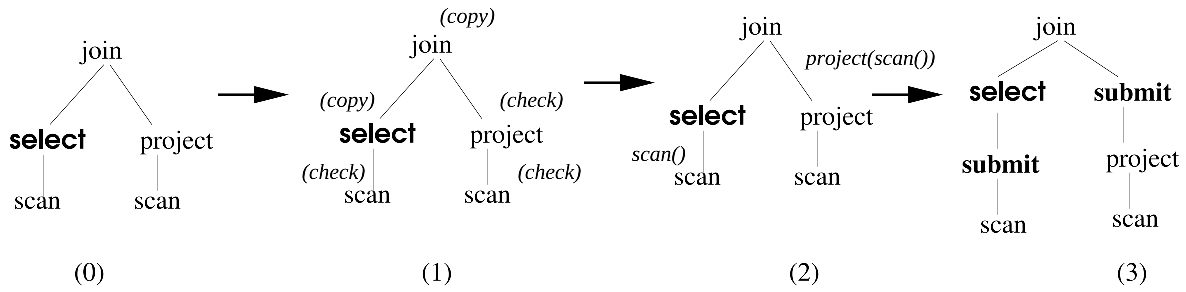


Figure 7: Example of applying `maxTreeExtended`. Tree (0) is an input tree. Operator `select` can not be executed by the source. Tree (1) is the result of bottom-up traversal of the algorithm. All nodes are marked and the operator `submit` is not introduced. Tree (2) shows the strings that are parsed on the top-down traversal of the algorithm. Assuming that the source allows the composition of `project` and `scan` as well as sub-queries consisting of `scan` alone, tree (3) is the result tree.

### 5.5 Discussion

The introduction of the composition grammar produces a complication for the query processor – it may not be possible to generate a tree that can process a query on a given wrapper. The algorithms of Section 4 avoided this by requiring that every wrapper process `scan`, but the possibility to specify composition grammar lifts this restriction. For some restricted query languages and algebras, it is possible to determine if such a tree can be generated or not. However in the general case, the problem remains open.

We have considered combining the operator description language and the operator composition language. However, using the operator composition language implies a more expensive algorithm for the generation of final logical operator trees. Since this algorithm is used extensively in query optimization, we have left the division of the two languages in order to study the impact of each language on query optimization in future work.

## 6 Implementation

This section describes the current implementation status of this work, in the framework of the Disco prototype [TAB$^+$]. The prototype is implemented in Java and communication with the wrappers is done using the Java Remote Method Invocation (RMI) mechanism.

**Logical Operators.** We have implemented the following logical operators at the mediator level: $UA_{med}$ = {`scan, select, project, join`}. These operators are those of relational algebra. The mediator also implements physical algorithms for the local execution of the composition query.

**Query Processor.** Currently, Disco implements a simple query processor. As part of the Disco query processor, we have implemented the extended version of `allTrees` and `maxTree`. We are currently implementing the `maxTreeExtended` algorithm. The choice of the algorithm, i.e., exhaustive tree generation, or the use of the heuristic, is a parameter of the query processor. This implementation was useful to validate our algorithms: the system generates *only* the logical operator trees that the wrapper can execute. `maxTree` generates the maximal possible tree, and `allTrees` generates all possible combinations of executable trees. We have added the ability to change the functionalities exported by the wrappers to experiment with the different combinations of functionalities. The examples in Section 4 are based on this experimentation work.

**Communication with the wrappers.** The mediator is currently connected with three wrappers implemented in the framework of the *Electronic Marketplace* project [BAE97], done between INRIA and Bull in the context of the Dyade joint venture. The goal of this project is to provide a uniform view of the heterogeneous distributed information sources in the domain of public construction. The sources are distributed over the Internet in France.

The mediator is a resource executing under the Jigsaw HTTP server software [BS97]. The wrappers are Java RMI servers that contact data sources. Two data sources are files, and the third one is a WWW server. Thus, in response to a sub-query, the wrappers for the file data sources read the associated files, and the wrapper associated with the WWW server reads the available HTML files, parses them and generates the appropriate answer. Communication between the mediator and the wrappers is accomplished using RMI, and the communication between a wrapper and the WWW server is via HTTP.

The wrappers are the following. The first wrapper provides access to the calls for tenders for public markets stored in the Lotus Database and presented on-line as the WWW site. The second wrapper allows access to a second source of the calls for tenders. Data is stored in a file. Both wrappers export the same schemas that describe their structure. A call for tender concerns a `market` in a `district`; it is issued on a `appearance` date and is valid until a `validity` date. The third wrapper provides the information about building and construction companies in France. The data is stored in a file. The schema exported by the wrapper consists of the company's `name`, its `activity`, its `address`, `phone` and `fax` numbers and the `district` the company is located in.

All three wrappers can perform `select` and `scan` operators that can be composed. The wrappers export their functionality in the form described in Section 5. The wrappers functionality descriptions are later used by the query processor.

The typical queries that we execute in this environment are the following:

- find the appearance dates of the calls for tenders in the given district,
- find the name of the companies and the market of the calls for tenders that are both located in the same district,
- find the name and the activity of the companies and the market of the calls for tenders that are both located in a given district, etc.

# 7 Related Work

All systems striving to integrate heterogeneous information sources face the problem of integrating different functionalities of local sources. We study the related work with respect to two questions: (i) how do existing systems describe source capabilities; (ii) how is this description used during the query processing. The approaches taken differ on the level at which the necessary check of source capabilities is done. Some systems (e.g. IRO-DB [GST96], Pegasus [DS95]) eliminate the problem during the schema integration process. Database administrators define a federated schema that is constituted from local schemas, representing local sources. The functions provided on the federated level are only those that can actually be performed on local data sources and queries containing functions that cannot be directly mapped on some local functions are not accepted.

Other systems integrate different domains but require a user to address each specific function in a domain explicitly. For example, Hermes [ACPS96] can integrate relational, object-oriented and spatial databases. Each domain is viewed as consisting of three components: a set of values $\Sigma$; a set $F$ of functions on $\Sigma$; and a set of relations on the data objects in $\Sigma$. The functions in $F$ take objects in $\Sigma$ as input and return as output objects from their range. This different domain knowledge is not "unified" at the global level, and each specific function in each domain has to be addressed explicitly.

Other systems prefer to detect the functional discrepancies during query processing by means of some kind of source description. InterViso [THMVB95] is a DBMS front-end that allows a user to access data managed by different heterogeneous distributed DBMSs. User query is formulated in terms of a federated schema and during query processing, this query is reorganized into the series of SQL statements that are directed to the local bases. All data operations are performed by local relational engines. To specify which relational operations can be performed at a data source, InterViso defines a *capabilities table* that includes the number of joins that can be done in one query, the availability of GROUP BY, aggregates, LIKE pattern matches, string operations, INDEX, and DISTINCT. If a data source cannot perform the needed function, the query is directed to another data source, or a missing functionality is supplied by a *schema translator*.

The local capabilities specified by the system concern mostly the different "dialects" of SQL and assume that the basic relational operators can be executed at a data source. The system is closely related to SQL query processing and cannot be easily extended to support various "operator sets".

Reference [LSK95] gives a language for describing information sources that reflect the content of those sources. The query generation algorithm operates in two phases - first it determines the join order, and then finds the relevant external sources to answer each conjunct of a query. The optimization criteria minimizes the number of external information sources accessed. In fact, source capabilities are modeled as a subset of queries the source is willing to answer and are presented like *query forms*. Nevertheless the plan generation algorithm supposes that each source is able to answer any query, so the source capabilities descriptions are not used.

Reference [LRO96] extends the language proposed in [LSK95] and creates a capability record for each source relation. These descriptions are of the form $(S_{in}, S_{out}, S_{sel}, min, max)$. $S_{in}$ is a set of attributes required on input, $S_{out}$ is a set of attributes returned from a source relation and $S_{sel}$ is a set of attributes on which a source can perform selection. A query plan is a sequence of accesses to the source relations (satisfying input requirements of each source) interspersed with the operations that are performed in the mediator. The query processor first generates semantically correct conjunctive plans and then orders the conjuncts (subgoals) of each plan to ensure that the plan is executable (which means that the input requirements of each source are satisfied). Reference [PGH96] generalizes source capability records.

Our operator functionality language is also a generalization of capability records. In addition, we provide the operator composition functionality and show how these languages can be used for an operator-based query processor.

The Garlic project [KHWY97] has an optimizer which exploits the knowledge of wrapper query capabilities. Their optimizer is based on use of grammar-like rules [Loh88]. Each wrapper exports its own set of rules that describes its own optimization. During optimization the optimizer "switches" from generic optimization rules to the rules specific to a wrapper. This work is closely related to ours. The main difference is that Garlic wrapper capability rules describe optimization and are necessarily intertwined with a particular optimizer, as opposed to our method of adding a step to a generic optimization framework.

In the distributed heterogeneous systems, the original query expressed in mediator terms is not directly executable on local sources. The related research area concentrates on formulating a semantically correct execution plan. Reference [LRO96] and [LSK95] show that the problem of finding a semantically correct query plans amounts to finding a conjunctive query that uses only local source relations and is *contained* in a given original query, and therefore is closely related to the problem of answering queries using views. Interested readers might consult [LMSS95, Qia96] that treat the problem of answering queries using a finite set of views and [LRU96] that considers the case of an infinite set of views.

# 8 Conclusion

A common architecture to integrate large numbers of distributed heterogeneous data sources consists of mediators that give a global view of the data sources and wrappers that give a local view of each source. Processing queries in this architecture is difficult because data sources have varying levels of functionality.

In Disco we deal with this problem by providing a flexible wrapper interface in terms of logical operators. When the DBI implements a new wrapper (for a new type of data source), she chooses a (sub) set of logical operators to support. This provides a good balance between implementation of new complex interfaces and the gain from additional complexity. To produce code for wrappers of varying functionality, the mediator incorporates the wrapper capabilities into query processing.

The novel contributions of this paper are the following. First, we have defined a composable operator model as wrapper interface. This model is simple and general enough to capture a wide range of data source capabilities.

Second, this interface is the basis for two flexible languages to specify the functionality of wrappers. The first interface language is used to describe the operators supported by a wrapper in fine detail, yet at a high-level of abstraction. This language eases the task of integrating a new wrapper. The second language is used to express the restrictions on operator composition imposed by a wrapper. The flexibility of our language permits also incremental wrapper implementation (going from simple operators to more complex ones).

Third, we proposed two algorithms for combining wrapper functionality with mediator query processing. The first algorithm generates all possible logical plans executable by a wrapper. The second one generates one plan that trades optimization time for query execution time using a heuristic. These algorithms work with any query processing algorithm that can accommodate logical operators.

Finally, we have described the current implementation of the operator model and query processing algorithm in the Disco prototype. The prototype is implemented as Java classes and includes wrappers for an existing application of Electronic Marketplace, which we develop with Bull.

Future work in this area will extend the wrapper interface language to increase the number of details in the wrapper functionality description. For instance, we will add the capability of describing complex predicates and object-oriented features (e.g., path expressions, etc.). Secondly, we will study the impact of our algorithms to the performance of the query optimizer. Finally, we will apply our language to other logical operators (e.g. those of geographical information systems) in order to incorporate specialized sources into our system.

### Acknowledgments

The authors wish to thanks Rémy Amouroux, Francoise Fabret and Philippe Bonnet for comments on earlier drafts of this paper.

## References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[BAE97] Electronic marketplace. http://www.multimedia.bull.net/info/bmm/bae/, 1997.

[BS97] A. Baird-Smith. Jigsaw HTTP server software and related activity. http://www.w3.org/pub/WWW/Jigsaw/Activity.html, 1997.

[DS95] W. Du and M-C. Shan. Query processing in pegasus. In *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. 1995.

[FRV96] D. Florescu, L. Raschid, and P. Valduriez. Query modification in multidatabase systems. *Int. Journal of Intelligent and Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems: Heterogeneous Databases*, 5(4), December 1996.

[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 Anniversary Meeting of Information Processing Society of Japan*, 1994.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GST96] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *Int. Conf. on Very Large Data Bases*, 1996.

[KHWY97] D. Kossmann, L. M. Hass, E. L. Wimmers, and J. Yang. I can do that! using wrapper input for query optimization in heterogeneous middleware systems. Submitted for publication, 1997.

[KLSS95] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*, 1995.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Int. Conf. on Principles of Database Systems*, 1995.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *ACM SIGMOD Int. Conf. on Management of Data*, 1988.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Int. Conf. on Very Large Data Bases*, 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external query processors. In *Int. Conf. on Principles of Database Systems*, 1996.

[LSK95] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5(2), September 1995.

[NGT97] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources, 1997. Submitted for publication.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Int. Conf. on Deductive and Object-Oriented Databases*, 1995.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *IEEE Int. Conf. on Parallel and Distributed Information Systems*, 1996.

[Qia96] X. Qian. Query folding. In *IEEE Int. Conf. on Data Engineering*, 1996.

[RAH+96] M. Roth, M. Arya, L. Haas, M. Carey, W. Cody, R. Fagin, P. Schwarz, J. Thomas, and E. Wimmers. The Garlic project. In *ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[TAB+] A. Tomasic, R. Amouroux, P. Bonnet, O. Kapitskaia, H. Naacke, and L. Raschid. The distributed information search component (DISCO) and the World-Wide Web. In *ACM SIGMOD Int. Conf. on Management of Data*. Prototype Demonstration. To appear., 1997.

[THMVB95] M. Templeton, H. Henley, E. Maros, and D. Van Buer. InterViso: Dealing with the complexity of federated database access. *VLDB Journal*, 4:287–317, 1995.

[TRV96] A. Tomasic, L. Rachid, and P. Valduriez. Techniques for scaling access to distributed heterogeneous databases in DISCO. In *Int. Conf. on Distributed Computing Systems*, 1996.

# A Operator Description Language BNF

Figure 8 gives the BNF of the operator description language described in Section 5

| | | |
|---|---|---|
| *functionalities* | ::= | (*operators*)* |
| *operators* | ::= | `operator_name` [ *collections* ] |
| *collections* | ::= | ( `collection_name` `MIN` { *collection* } )+ |
| | | \| ALL |
| *collection* | ::= | ( *binding* `attribute_name` ( *predicates* ))+ |
| | | \| ALL |
| *predicates* | ::= | ( `predicate_name` )* |
| | | \| ALL |
| *binding* | ::= | [bind] [combine] |

Figure 8: BNF of the operator description language

The non-terminals of the grammar are given in *this* font. (e)* means that e can be repeated zero times or more, (e)+ means that e can be repeated once or more. Square brackets [...] indicate that the ... is optional (here, both bind and combine are optional). Keywords are given in this font. The language has the following keywords: bind, combine, ALL. The keyword bind before the attribute_name means that the given attribute should be bound on input, i.e., it must appear in the attribute for this operator. The keyword combine before the attribute_name means that the given attribute can be bound on input at the same time as some other attribute. The keyword ALL can replace (i) the list of all collections on which operator can be executed, or (ii) the list of attribute names, or (iii) the list of the predicates supported. The tokens given in `this` font are the terminals of the grammar. The language uses the following terminals: `operator_name`, `collection_name`, `attribute_name` and `predicate_name`. The values of these terminals are known to the system at the connect time. `MIN` is an integer number indicating the minimal number of attributes that should be bound on input.

# B Algorithm `maxTreeExtended`

This appendix extends the algorithm `maxTree` with the operator composition order verification. The description of this algorithm is given in Section 5.3.

Note, that the result of the `bottomUp` function is an *annotated* logical operator tree, i.e., a tree where each node consists of a pair of logical operator and its mark. This annotated tree is given as input to the `topDown` function.

```
LogicalOperator maxTreeExtended(Wrapper w, LogicalOperator tree, Set capabilities) {
    (newtree, mark) = bottomUp(tree, capabilities);
    result = topDown(w, (newtree, mark));
    // verify if each path in result has submit
    if(verify(result))
        return result;
```

```
    else
        return NULL;
}



// check=false, copy=true
(LogicalOperator, boolean) bottomUp(LogicalOperator tree, Set capabilities){
    op = tree.topOperator();
    if (op.children == 0) // must be a scan and must be in capabilities
        return (tree, check);
    else
        b = false;
        S = ∅;
        foreach (i ∈ op.children())
            (Set_i, mark) = bottomUp(i, capabilities);
            S = S ∪ (Set_i, mark)
            b = b ∪ mark;
        if (!b)
            if (op ∈ capabilities)
                return (tree, check)
            else
                return (tree, copy)
        else
            return (new Operator(op, S), copy)
}
```

```
LogicalOperator topDown ( Wrapper w, (LogicalOperator tree, boolean mark) ) {
    op = tree.topOperator();
    if (mark = check ) {
        boolean goodOrder = w.parse(tree.toString());
        if (goodOrder)
            return new Submit(tree);
    }
    if (op.children() == 0) {
        return tree;
    }
    else {
        S = ∅;
        foreach (i ∈ op.children() )
            S = S ∪ topDown(w, (i, mark));
        return new Op(S, op);
    }
}
```

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