Defining the search space for query optimization in a heterogeneous database management system [*]

Daniela Florescu[**], Louiqa Raschid[†], Patrick Valduriez[**]

[**]INRIA Rocquencourt, Projet RODIN
BP 105, 78153 Le Chesnay Cedex, France
firstname.lastname@inria.fr

[†]College of Business and Management and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
louiqa@umiacs.umd.edu

November 9, 1994

Abstract:
We consider a heterogeneous database system (HDBMS) with a global object schema, and remote target databases that may be relational or object-oriented. The problem is to transform an initial query, posed against the global object schema in an object query language, into simplified queries that can be efficiently processed against some remote target databases. In the HDBMS, query optimization is made difficult by schematic discrepancy, and the need to model mapping information between the global and target schema(s). In this paper, we address this problem by representing the mappings from a target schema to the global schema, as a set of heterogeneous object equivalences, in an algebraic language. These equivalences are the basis for defining the heterogeneous search space of an HDBMS optimizer. We extend a modular rule-based query optimizer to produce simplified optimized queries for the HDBMS. The main result, reported in this paper, is the ability of the optimizer to uniformly perform syntactic simplifications, logical and semantic equivalence transformations, as well as heterogeneous equivalence transformations.

[*]This research has been partially supported by the Advanced Research Project Agency under grant ARPA/ONR 92-J1929 and by the Commission of European Communities under Esprit project IDEA.

1 Introduction

Research in heterogeneous database systems (HDBMS) has focused on resolving dissimilarities due to mismatch in data models, schema and query languages. Queries are executed against some target databases, so that information from these databases is accessible to other applications. In order to successfully support seamless integration of data from multiple sources, these systems must also provide heterogeneous query optimization, so queries can be processed efficiently. In this paper, we describe our research on extending a modular rule-based query optimizer [Florescu and Valduriez (1994)], to produce optimized queries in a heterogeneous environment.

We define heterogeneous query optimization for the purposes of this paper, as follows: Assume a global (object) schema in which a query is being posed in an object query language. For this paper, we use the Flora object model and Flora query language [Novak *et al* (1994)], developed at INRIA, as our global data model and query language. The queries are to be processed against some remote target databases, which could be represented by relational or object schemas. We assume there is knowledge of the mappings needed to *import* data from these remote databases. The objective of heterogeneous query optimization is to start with an initial query, say a Flora query, and perform possible simplifications and equivalent transformations to first produce (multiple) query (queries) in the Flora query language, which refer(s) to the remote schema entities. We could subsequently transform these queries and generate a query in the remote query language, or generate multiple sub-queries, if the information spans more than one remote database. We note that although we assume a global schema architecture for HDBMS, in this paper, we can apply our results to define a heterogeneous search space for a different architecture, e.g., federated HDBMS [Sheth and Larson (1990)], or canonical HDBMS [Qian and Raschid (1995)].

The role of an optimizer is to determine an execution plan that minimizes an objective cost function. The problem can be described abstractly as follows [Du92]: Given a query Q, an execution space E that computes Q, and a cost function c defined over E, find an execution e in E_Q (the subset of E that computes Q) of minimum cost: $[min_{\ e \in E_Q}\ \ c\ (e)]$.
An optimizer can be described along the following three dimensions: an *execution space* which captures the underlying execution model and defines, in an abstract way, the alternative executions; a *cost model* which predicts the cost of an execution; and a *search strategy* which is used to enumerate the executions and select the best one. The input query is often in an algebraic form, and execution plans are represented by algebraic operator trees. The execution space for a given query is obtained by rewriting algebraic expressions using equivalence rules. In this paper, we focus on the first task of defining the heterogeneous search space, in terms of an algebraic representation.

There have been several proposals for transforming queries in heterogeneous environments. Some of them are based on higher-order query languages, higher-order logics, or meta-models [Barsalou and Gangopadhyay (1992), Krishnamurthy *et al* (1991), Lakshmanan (1993)], and are not well supported under current DBMS technology. There have been a few systems that have studied query processing and/or have been implemented, e.g., Multibase [Dayal and Hwang (1984)], Pegasus [Ahmed *et al* (1991), Albert *et al* (1993)], UniSQL/M [Kim *et al* (1993), Kim and Seo (1992)], SIMS [Arens and Knoblock (1992), Arens *et al* (1993)]. Alternative approaches either specify a canonical mediator knowledge base which has mapping knowledge among different schema [Chang *et al* (1994), Qian (1993), Qian and Raschid (1995), Raschid *et al* (1993), Raschid and Chang (1994)], or advocate the federated approach, as summarized in [Sheth and Larson (1990)]. There has also been research on constructing a calibrating database, to develop cost-models which capture the various parameters corresponding to HDBMS [Du *et al* (1992)]. This study assumes that the first step of resolving discrepancies among the schema has already been solved.

The most popular approach is to build a single unified global schema, to resolve all the schematic discrepancies and representational mismatch (due to different data models), of the heterogeneous environment. Each of the remote schemas are integrated within the global schema, and each of the remote databases is assumed to be a model for the global schema. The mapping from each remote schema to the global schema is often expressed in some high-level extended SQL-like data definition/manipulation language. For example, HOSQL in the case of Pegasus and SQL/M in the case of UniSQL/M.

There has been some research in query optimization and query decomposition in heterogeneous environments in the Pegasus, SIMS and UniSQL/M projects. For example, the Pegasus project [Ahmed *et al* (1991), Albert *et al* (1993)], studies both query reformulation as well as query optimization. During query processing, the HOSQL query is represented as an E-tree (of sub-queries and operations), in some extended relational algebra. If there are multiple external sources for the data (EDRM), then appropriate translations are performed. The query in the native query language is represented in some canonical form, and then decomposed into parametric sub-queries, each evaluated in a separate EDRM. Some simplification of the queries is also possible, e.g. elimination of the expensive outer-join operation. They also consider some cost-based and heuristic optimization. However, these strategies are applied ad-hoc, and they do not define a heterogeneous search space.

The SIMS project [Arens and Knoblock (1992), Arens *et al* (1993)], also addresses the issue of query processing and optimization. However, they use the LOOM knowledge representation system to build a global schema, and as their query language. All optimization is done within the LOOM reasoning module, and not in an algebraic representation. Thus, it is difficult to determine if they systematically search the space of equivalent queries. A declarative specification of mappings among multidatabase systems is presented in [Chomicki and Litwin (1994)], but they do not address the issue of generating equivalent queries using this knowledge.

To summarize, previous research does not systematically define a search space for heterogeneous query optimization. A systematic definition of this space is the only way to exploit effective query optimization techniques (together with a cost model and search strategy), for HDBMS. This will become more important, as HDBMS scale up to large numbers of remote databases.

Our approach to heterogeneous query optimization is influenced by the modular rule-based Flora object query optimizer [Florescu and Valduriez (1994]. We define a heterogeneous search space for generating equivalent queries, in which we uniformly apply transformations based on syntactic simplification, logical and semantic equivalences, and most importantly, we incorporate heterogeneous equivalences among entities in multiple heterogeneous schema. We specify "heterogeneous object equivalences" to map from the entities of a remote target schema to the unified global schema, using a second-order algebraic language [Florescu and Valduriez (1994), Guting (1993)]. The heterogeneous search space is then defined using heterogeneous object equivalences, and logical and semantic equivalences in both the global and target schema(s).

Our approach to specifying heterogeneous equivalences is similar to the concept of virtual classes, proposed in [Abiteboul and Bonner (1991)]. A virtual class is populated by "copying" objects already existing in other classes, or by "creating" entirely new objects. When a class is populated by already existing objects of another class, they offer the possibility of both importing the class (definition), and hiding portions of it; this is similar to a view. However, they do not consider objects in a virtual class referencing objects in some other class, possibly in a recursive fashion. This is an important extension that we have considered in our research.

Through a suite of examples, we explore the heterogeneous search space for query optimization. We summarize the examples as follows: The optimizer obtains a simplified query wrt the remote schema, through the uniform application of heterogeneous equivalences, together with syntactic simplification (such as un-nesting), logical simplification (based on algebraic properties of built-in operators in the algebra, e.g., membership test), and semantic equivalences (based on domain knowledge specified as integrity constraints). Even with redundancy of heterogeneous object equivalences, the optimizer correctly produces a simplified query. However, the optimizer can also use alternative heterogeneous and semantic equivalences to produce several candidate equivalent queries, from which the optimal query must be chosen. When remote schema entities are located in multiple remote databases, the optimizer can produce equivalent queries, as well as sub-queries.

This paper is organized as follows: In Section 2, we give an overview of the Flora object model and Flora query language, and the modular rule-based query optimizer architecture. In Section 3, we define "heterogeneous object equivalences", and give examples. We also define the heterogeneous search space for the HDBMS. In section 4, we use a suite of example schemas and queries, to illustrate the simplification process, and to describe alternate queries in the heterogeneous search space. Finally, in Section 5, we summarize our results, and discuss future research.

2 The Flora query optimizer

Flora is a simple algebraic language [Novak *et al* (1994)], for an object data model, with collection-oriented computational capabilities. The Flora query optimizer uniformly captures logical, semantic, and implementation knowledge (physical information, statistics, etc.), within the framework of a many-sorted algebra, and uses this knowledge to find an efficient execution plan, from among a set of alternative plans. The optimizer is modular and rule-based, and strives to balance extensibility with efficiency.

2.1 The Flora Algebra

The framework is based on a many-sorted algebra, which, unlike the relational algebra, supports arbitrary types, and operations between arbitrary types. A formal description of the algebraic representation is in [Florescu and Valduriez (1994), Guting (1993)].

Flora type expressions are constructed from the basic types (atomic built-in types, external types, and user-defined object types), through the recursive application of the following type constructors: set, list and tuple.

The set of corresponding Flora operators include built-in and user-defined operators. The built-in operators include comparisons, logic and arithmetic operators, aggregation operators, the set membership operator, etc. A definition of built-in Flora types and operators is in appendix A.

The part of the algebra specific to an application domain is described by an object signature $(\mathcal{O}, \sum_{\mathcal{O}}, \prec, \mathcal{I})$, where $\mathcal{O}$ is a set of object type names, $\sum_{\mathcal{O}}$ is a set of user-defined operators (with their signatures), $\prec$ is a partial order on $\mathcal{O}$ (corresponding to the inheritance relationship), and I is a function that associates every object $o \in \mathcal{O}$ with a Flora type expression, (corresponding to the state of the object type o).

Example 2.1

Consider the following object signature: $\mathcal{O}$={**Person**, **City**, **Country**}, $\sum_{\mathcal{O}}$={}, $\prec$ ={} and
I(**Country**)=[name: **string**, cities: {**City**}, capital: **City**]
I(**City**)=[name: **string**, residents: {**Person**}, country: **Country**]
I(**Person**)=[name: **string**, country: **Country**, city: **City**, income: **int**, child: {**Person**}].
NV is a set of particular named variables, each associated with a Flora type. For every object type $o \in \mathcal{O}$, there exists an implicit named variable ext(o) of type $\{o\}$ corresponding to the extension of this object type. The value of the variable ext(o) is the set of object identifiers of all the persistent instances of type o, (including instances of all sub-types of type o).

Example 2.2

NV = { ext(Country) of type {**Country**}, ext(City) of type {**City**}, ext(Person) of type {**Person**}, RICH of type **int** }.

A Flora term is an arbitrary combination of user-defined and built-in functions, starting with constants and variables. X is a set of variables, each associated with a type, and $T(X)$ is the set of well-typed terms. A formal definition of Flora well-typed expressions is in appendix B.

The most important Flora built-in operator is **Select**. It provides a simple but expressive way of constructing Flora queries, and it is a good candidate for optimization. **Select** generalizes the classic select-project-join relational operators. It has three arguments, corresponding to the SQL clauses, SELECT, FROM and WHERE.

Definition 2.1 Select:
Let $C_1 \in T(X)$ *be of type* $\{s_1\}$, $C_2 \in T(X \bigcup \{x_1\})$ *be of type* $\{s_2\}$, $\cdots$, $C_n \in T(X \bigcup \{x_1, \cdots, x_{n-1}\})$

be of type $\{s_n\}$*, pred*$\in T(X \bigcup \{x_1, \cdots, x_n\})$ *be of type bool, and proj*$\in T(X \bigcup \{x_1, \cdots, x_n\})$ *be of type* s*. Assume that* $\{x_1, \cdots, x_n\}$ *and* X *are disjoint. Then,*

$$\textbf{Select}(proj,\ x_1\ in\ C_1\ and\ x_2\ in\ C_2 \cdots x_n\ in\ C_n,\ pred)$$

is in $T(X)$*, and it is of type* $\{s\}$ $(n \geq 1)$*.*

Queries can be expressed wrt any set-oriented expression, including the extensions of the classes. A range variable x_i is associated with every collection C_i, x_i is of type s_i and C_i is of type $\{s_i\}$. (See [Florescu and Valduriez (1994)] for details.) The expressions for each C_i, *pred* and *proj* are Flora terms, which satisfy the type constraints. Flora allows nesting and user-defined functions in all clauses of the **Select**, as illustrated in the following simple examples:

Example 2.3

Flora queries

a) Select the names of rich people who are childless.

```
select(x.name, x in ext(Person),
          x.income>RICH and is_empty(x.child))
```

b) Select the names of people who have at least one child residing in Paris.

```
select(x.name, x in ext(Person),
          exists y in x.child (y.city=Paris));
```

Object query languages may support the creation of new objects [see Bancilhon *et al* (1989)]. For our research, this feature is crucial, since new objects are created in the heterogeneous environment, when a query is executed in a remote database.

Definition 2.2 *Object construction:*
Let $o \in \mathcal{O}$ *be an object type name and* $t \in T(X)$ *be of type* $I(o)$*. Then,* **NewObject** *o(t)* $\in T(X)$*, and it is of type o.*

Example 2.4

Let i_1, i_2, i_3 and i_4 be object identifiers corresponding to an instance of type **Country**, an instance of type **City**, and two instances of type **Person**, respectively. Then,
`NewObject Person ([name:="Paul", country:=`i_1`, city:=`i_2`, income:=20000, child:={`i_3, i_4`}])`
creates a new object Paul of type Person.

2.2 The search space for a Flora query

We define a query plan to be a *term of an object signature*, over the set of named variables. The task of an optimizer is to find an efficient query plan, equivalent to an input query plan. We say that two terms t_1 and t_2 of an object signature are equivalent, (noted as $t_1 \equiv t_2$), if they evaluate to the same value, for each instantiation of the named variables. Given an input term, t, over named variables NV, representing an input query plan, we define $Equiv(t) = \{t' \mid t' \in T(NV)$ and $t' \equiv t\}$, to be the search space for the Flora query t, that is being explored by the optimizer.

Defining the equivalence of terms is an important task. For the relational algebra, the equivalence of terms is based on the well-known algebraic properties of the relational operators. There are also well-known logical equivalences for built-in operators. However, the application dependent object signature introduces new types and operators, into the object algebra, for each application. The object query optimizer must be capable of incorporating this application specific knowledge. Thus, semantic information, or knowledge about the object signature, for an application, is expressed declaratively, as integrity constraints. They are specified as *object equivalences*, and define the correct transformations between equivalent algebraic terms with variables. Object equivalences are defined as follows:

Definition 2.3 *Assume an object signature* $(\mathcal{O}, \sum_{\mathcal{O}}, \prec, \mathcal{I})$ *and a set of named variables* NV.

- *Let* X *be a set of variables which are disjoint from* NV. *Let* E_1, $E_2 \in T(NV \cup X)$ *be two well typed terms using variables* $X \bigcup NV$ *having the same type. Then, the triple* (X, E_1, E_2) *is an object equivalence. If* $X = \{x_1, \cdots, x_1\}$ *and* x_i *has the type* s_i *for* $i=1,\cdots,n$, *then the object equivalence is the following first order formula:*

$$\forall x_1 \text{ of type } s_1, \cdots, \forall x_n \text{ of type } s_n \qquad (E_1 \sim E_2).$$

- *The equivalence* (X, E_1, E_2) *is valid in the algebra, if, for all instantiation* ϕ *for variables* $NV \cup X$, *the result of the evaluation of term* E_1 *is identical to the result of the evaluation of term* E_2.

Example 2.5

The following are example integrity constraints that are specified as object equivalences, for the object schema in example 2.1:

a) There is a 1-to-many inverse association between instances of `Country` and `City`, via `residents` and `city`, as follows:

$\forall$ `x in ext(Country)` $\forall$ `y in ext(City)    IsIn(x.cities, y)` $\sim$ `x.country = y`

b) The capital of a country must also be a city of this country.

$\forall$ `x in ext(Country)    IsIn(x.cities, x.capital)` $\sim$ `TRUE`

An optimizer uses object equivalences as rewriting rules, to generate equivalent plans.[1]:

[1]A type-based pattern-matching algorithm, used by the optimizer, is described in [Florescu and Valduriez (1994)]

Definition 2.4 *Given a valid object equivalence r=(X, E_1, E_2), and a query $t \in T(NV)$, then $t \equiv t'$, where $t' \in T(NV)$, if there is a subexpression t'' of t, and a substitution θ for the variables X, such that t''=$E_1\theta$ and t' is obtained by replacing t'' in t by $E_2\theta$; t' is an equivalent query for t.*

2.3 The Flora optimizer architecture

The Flora optimizer architecture is described in detail in [Florescu and Valduriez, (1994)]. A critical design decision was to use (declarative) semantic information together with hard-coded transformations in a uniform manner, in generating the search space. This is supported by the modular nature of the optimizer architecture, which strikes a balance between extensibility (allowing the addition of new transformation rules), and efficiency (by hard-coding some transformation rules).

Knowledge that is described declaratively include the algebraic properties of built-in operators and user-defined operators, (e.g., commutativity of the union), integrity constraints for each object schema (e.g., inverse links), and the correspondence between logical operators and physical operators that implement them. (We do not discuss this last category in the context of HDBMS.)

The Flora optimizer is composed of modules, within a hierarchy. Each module has a specific goal and a strategy, which could include invoking another module. For each query plan that is input, a module produces one or several equivalent query plans. Major modules are query simplification (which include some hard-coded transformations such as un-nesting, quantifier elimination, etc.), and logical transformation (using logical equivalences and semantic equivalences). Other optimizer modules such as operator ordering are not discussed in the context of the HDBMS.

3 Heterogeneous object equivalences and the search space

Just as object equivalences were used to incorporate application specific domain knowledge, we use *heterogeneous object equivalences* to specify mapping knowledge between entities in the global object schema, and remote entities in the target schema, (which may be relational or object schema). These heterogeneous equivalences are to be utilized during optimization. Normally, we do not attempt to share actual object identifiers across databases in a HDBMS, and any links between entities must necessarily be value-based. The heterogeneous equivalences express the link between an object instance that is created in the global schema, and the "values" corresponding to this object, which are imported from the target databases. We first present an example of a simple heterogeneous object equivalence. Then, we present a formal definition.

Example 3.1

```
Global object schema                                  Remote relational schema

Class globalPerson                                      remPerson(Pname, Page, Pssn)
   { name: string, age: integer, ssn: string}
```

In this example, there is a class `globalPerson` in the global schema, and a corresponding relation `remPerson` in the target relational schema. We express a heterogeneous equivalence in a straightforward manner, to reflect that a tuple of the relation `remPerson` corresponds to an instance of the class `globalPerson`. The equivalence is as follows:

```
ext(globalPerson) ~ Select(NewObject([ name := x.Pname,
                                       age := x.Page,
                                       ssn := x.Pssn ], x in remPerson) )
```

We now define the heterogeneous object equivalence formally. We use a *heterogeneous object signature* to describe the HDBMS environment, consisting of the global schema and all remote schema(s).

Definition 3.1 *Given a global object signature* $(\mathcal{O}^g, \sum_{\mathcal{O}^g}, \prec^g, I^g)$, *and a set of n remote object signatures, (corresponding to n remote schema(s),* $(\mathcal{O}^r_i, \sum_{\mathcal{O}^r_i}, \prec^r_i, I^r_i)$, $i = 1, n$, *then, a heterogeneous object signature is* $(\mathcal{O}^h, \sum_{\mathcal{O}^h}, \prec^h, I^h)$, *where*
$\mathcal{O}^h = \mathcal{O}^g \bigcup (\bigcup_{i=1,n} \mathcal{O}^r_i)$, $\sum_{\mathcal{O}^h} = \sum_{\mathcal{O}^g} \bigcup (\bigcup_{i=1,n} \sum_{\mathcal{O}^r_i})$, $\prec^h = \prec^g \bigcup (\bigcup_{i=1,n} \prec^r_i)$, *and function* $I^h(o)$ *for an object of type* $o = \begin{cases} I^g(o) \text{ if } & o \in \mathcal{O}^g \\ I^r_i(o) \text{ if } & o \in \mathcal{O}^r_i \end{cases}$

We assume that the object type names, and the user-defined operator names,[2] in the global schema and the remote schema(s) are disjoint.

Definition 3.2 *Suppose* NV^g, NV^r_i, $i = 1, n$, *are the set of named variables, of the global schema, and the remote schema(s), respectively. Then, the set of named variables in the heterogeneous schema is* $NV^h = NV^g \bigcup (\bigcup_{i=1,n} NV^r_i)$.

In the case of a target relational schema, the corresponding object signature is $(\emptyset, \emptyset, \emptyset, \emptyset)$ and the named variables are the set of relation names.

We can now define a heterogeneous object equivalence, as follows:

Definition 3.3 *(***Heterogeneous object equivalence***)*
Suppose o is an object type name in the global schema, $C_1, C_2, \cdots, C_n \in NV^h$ *are named collections in the HDBMS (i.e., global and remote schema(s)), pred*$\in T(\{x_1, \cdots, x_n\})$ *and it is of type* **bool**, *and proj*$\in T(\{x_1, \cdots, x_n\})$ *and it is of type* $I^g(o)$. *Then*

$$ext(o) \sim \textbf{Select}(\textbf{NewObject}(proj),\ x_1 \text{ in } C_1 \text{ and } x_2 \text{ in } C_2 \text{ and } \cdots \text{ and } x_n \text{ in } C_n,\ pred)$$

is a heterogeneous object equivalence.

This definition specifies the extent of a class o, in the global schema, (similar to a view). Objects of the class o are created after importing data from the remote database(s). Each query on the object type o, in the global schema, referred to as `ext(o)`, will obtain instance(s) of some data

[2]In this paper, we do not consider any user-defined operators in the remote schema.

values, as specified in the `Select(NewObject(···), ··· )` definition. The query will create a new object for each appropriate (combination of) value(s). In this paper, we assume that there is some knowledge of a "key", with an implicit uniqueness constraint, which is used to correctly specify heterogeneous equivalences.

Next, we define the heterogeneous search space for the HDBMS environment. We build on a previous definition of the search space $Equiv(t)$, expressed wrt a single object schema. We define $SearchSpace(t)$, for the HDBMS, for input query t in the global object schema, as follows:

Definition 3.4 *Let t be a term of an object signature of the global object schema, $t \in T(NV^g)$. The heterogeneous search space for t is the set of equivalent queries, t', such that they are expressed only with named variables corresponding to the remote schema(s).*

$$SearchSpace(t) = \{t' \in T\ (\ \bigcup_{i=1,n} NV^r_i) \mid t \equiv t'\} \qquad \forall\ t \in T(NV^g)$$

We now address three important issues that affect heterogeneous equivalences in a HDBMS.

Object Reference: Consider an attribute of an object whose value refers to another object, in either the global schema, or in some remote DBMS. It could also be an imaginary object that is created. Such a situation is not considered in [Abiteboul and Bonner (1991)]. One possibility is to create a new object, for each reference to this object. However, if there were multiple references to this object, then this solution could lead to multiple objects (identifiers) corresponding to an identical set of values. This is a problem, since it is only on the basis of the "values" that we are able to identify equivalences in the HDBMS, and create new (imaginary) objects in the heterogeneous environment. The solution is to avoid the creation of new objects, in this context.

Recursive Classes: There is a related problem when we consider classes that are recursively defined. Suppose there is a class `globalPerson` in the global schema, and two relations in the remote schema, `remPerson` and `remMarried`, from which `globalPerson` instances are to be populated. Note that `spouse` refers to an instance of `globalPerson`, hence the recursive reference in the heterogeneous object equivalence, as follows:

Example 3.2

```
Global object schema                Remote relational schema


Class globalPerson                     remPerson(Pname, Page)
   { name: string                      remMarried(Pname1, Pname2)
     age: integer
     spouse: globalPerson }
```

We define the heterogeneous object equivalence recursively, as follows:

```
ext (globalPerson) ~
    Select(NewObject([ name := x.Pname
                       age := x.Page
spouse := Select y,y in ext(globalPerson) & z in remMarried
                   & ( (y.name = z.Pname1 & x.name = z.Pname2)
                     or (y.name = z.Pname2 & x.name = z.Pname1) ) ]
          x in remPerson )
```

However, these recursive references may give rise to another problem. Consider a query in the global schema which attempts to access objects (identifiers) from the remote schema, rather than some "descriptive value" associated with the objects. If there is a recursive heterogeneous equivalence defined for this object, then the query transformation algorithm will recursively apply the heterogeneous equivalences, and will not terminate. Consequently, we can only deal with queries that access "values" from remote databases, in HDBMS. Since we do not share object identifiers across databases in HDBMS, this solution will suffice for our purposes. However, this is a problem that merits further research [Raschid *et al* (1995)].

Referencing Existing Objects: So far we have expressed the heterogeneous equivalences in a straightforward manner, creating *new objects* for each combination of values obtained from some remote database(s), when a query is executed. This solution would be unsatisfactory if these objects are to be made persistent in the global environment, i.e., we would no longer be able to keep on creating new objects. A further drawback is that we are not able to identify already existing objects, for which there are existing object identifiers, in the global environment, but whose data values may only be available in some remote database(s). Some solutions were briefly discussed in [Abiteboul and Bonner (1991)], and these problems also merit further study.

4 Examples of queries in the heterogeneous search space

We describe the control strategy of the optimizer briefly, to generate equivalent queries in the heterogeneous search space. A Flora query, posed wrt the global schema entities, is first syntactically simplified. Next, any applicable logical transformations wrt global schema entities are applied. This includes both logical equivalences wrt built-in operators and semantic equivalences based on integrity constraints in the global schema. We then apply relevant heterogeneous equivalences, so that references to global schema entities are replaced with references to entities in the remote schema(s). Further, we apply any applicable logical transformations (based on semantic equivalences wrt the remote schema entities). In this paper, we do not consider logical equivalences for built-in operators in the remote schema(s). Since the application of the heterogeneous equivalences may introduce new references to entities in the global schema, we continue until all such references have been replaced. The process may produce one or more equivalent plans, wrt the remote schema(s). We also do not consider the control strategy for operator ordering, or for selecting a particular physical implementation for an operator, etc.

In this section we use a suite of examples to illustrate the process of simplifying queries and obtaining equivalent queries in the heterogeneous search space. We summarize the examples as follows: The optimizer obtains a simplified query wrt the remote schema through the uniform application of heterogeneous object equivalences, syntactic simplification, logical simplification, and semantic equivalences. Even with redundancy of heterogeneous object equivalences, the optimizer correctly produces a simplified query. However, the optimizer can also use alternative heterogeneous and semantic equivalences to produce several candidate equivalent queries, from which the optimal query must be chosen. When remote schema entities are located in multiple remote databases, the optimizer can produce equivalent queries, as well as sub-queries. If these equivalent queries refer to entities in multiple remote databases, then we decompose the queries. We use some simple syntactic analysis of the queries, and information on where the remote entities are located, to obtain simpler sub-queries, for each of the remote databases.

4.1 An example applying logical and heterogeneous equivalences

This example demonstrates the application of logical and heterogeneous equivalences, to produce a simplified equivalent query. Logical equivalences typically express some possible simplifications wrt a built-in operator. In this example, we consider both the cardinality operator *card*, and the membership operator *IsIn*, both of which are defined for sets.

In the example global schema, the class `Student` has an attribute courses referring to a set of `Course` instances, which are also in the global schema. This object reference is reflected in the heterogeneous equivalence rule H_1, defining `Student`, which refers to `RemStudent` and `RemEnrolled` in the remote schema, as well as to instances of `Course` in the global schema. There is a rule H_2, the heterogeneous equivalence corresponding to the class `Course` in the global schema. The schema and heterogeneous equivalences are as follows:

Example 4.1

```
Global schema                         Remote schema


Class  Student                          Relation RemEnrolled(name1, course1)
   { sname: string
     courses: set(Course) }             Relation RemStudent(name1)


Class Course
   { cname: string                      Relation RemCourse(course1, time1)
     time: float }
```

The heterogeneous equivalences are as follows:

```
[H1] ext(Student) ~
       Select(NewObject([sname:  x.name1,
                         courses:=Select(y,y in ext(Course) & z in RemEnrolled &
                                   z.name1 = x.name1 & z.course1 = y.cname) ],
                         x in RemStudent )
```

```
~ Select(NewObject( [sname:=···,
                     courses:=Select(y, y in ext(Course) & z in RemEnrolled
                     & PRED(x,y,z) ], x in RemStudent)
where PRED(x,y,z) is the predicate (z.name1=x.name1 & z.course1=y.cname)
```

```
[H2] ext(Course) ~ Select(NewObject([cname:=t.course1, time:=t.time1]), t in Course1 )
```

We next consider a logical equivalence rule corresponding to the Flora built-in cardinality operator, defined on sets, for example, card(X), where X is of type `set(Course)`. Recall that all references to a remote schema must be value-based. The following equivalence rule allows us to perform a simplification so that the cardinality operator is evaluated over a set of course names, for `set(Course)`, and eventually over the values of `course1` from the remote database. For simplicity, we present the particular form of the logical equivalence rule, as directly applicable to this query, as follows:

```
[L1] ∀ X: set(Course)   card(X) ~ card(Select(x.cname, x in X)
```

Consider a query to select the names of all students enrolled in exactly two courses. The corresponding Flora query is as follows:

```
Select(t.sname, t in ext(Student), card(t.courses) = 2)
```

We follow this query through the following steps, applying the equivalences H_1, L_1 and then H_2 in turn, as follows:
[Applying H_1 for `ext(Student)`]

```
Select(t.sname, t in Select(NewObject(
                               [sname:=x.name1,
                                courses:=Select (y, y in ext(Courses),
                                  z in RemEnrolled & PRED(x,y,z))],
                       x in RemStudent),
                       & card(t.courses) = 2)
```

[Simplifying for `t.sname` and applying L_1 to `card(t.courses)`]

```
Select( ((NewObject([sname:=x.name1, courses:= ... ])).sname, x in RemStudent,
             & card(Select(c.cname, c in t.courses))=2)
             & t in Select(Newobject(
                               [sname:=x.name1,
                                 courses:=Select (y, y in ext(Courses),
                                     z in RemEnrolled & PRED(x,y,z) )] ) ) )
```

[Simplifying for `t.courses`]

```
Select(x.name1, x in RemStudent,
        & card(Select(c.cname, c in
                          Select(NewObject[courses:=Select (y, y in ext(Courses),
                                 z in RemEnrolled & PRED(x,y,z)])).courses))=2)
```

[Simplifying `card(.....)`]

```
Select(x.name1, x in RemStudent,
        & card(y.cname, y in ext(Course) & z in RemEnrolled, & Pred(x,y,z) ))=2)
```

[Applying H_2 to `ext(Courses)`]

```
Select(x.name1, x in RemStudent,
          & card(y.cname, y in Select(NewObject(cname:=t.course1, time:=t.time1),
                     t in RemCourse)
          & z in RemEnrolled, & Pred(x,y,z) ) = 2)}
```

[Further simplifying]

```
Select(x.name1, x in RemStudent,
          & card(Select(t.course1, t in RemCourse & z in RemEnrolled &
                      z.name1=x.name1 & z.course1=t.course1) == 2)
```

The final simplified query only retrieves relevant information from the remote database. It also evaluates the cardinality operator over the values of `course1`.

Next, we consider a query that uses the expensive membership operator *IsIn*. We use a corresponding logical equivalence to eliminate this operation. The following query selects the names of all students enrolled in “cs101”:

```
Select(x.name, x in ext(Student), IsIn(Select(y.cname, y in x.courses), ‘‘cs101’’)
```

We first simplify the query as follows:

```
Select(x.name, x in ext(Student), y in ext(Course) &
             IsIn(x.courses, y) & y.cname = ‘‘cs101’’
```

We then apply equivalence H_1 to `ext(Student)` and simplify further, (for `x.name`) to obtain the following:

```
Select(t.name1, t in RemStudent, y in ext(Course), &
         IsIn(Select(z, z in ext(Course) & z’ in RemEnrolled & PRED(t,z,z’) ), y)
             & y.cname = ‘‘cs101’’
```

`where PRED(t,z,z’)` $\sim$ `(z’.name1=t.name1 & z’.course1=z.cname)`

We next apply the logical equivalence L_2 wrt the operator *IsIn*. This equivalence eliminates the membership test. For simplicity, we give the specific form of this equivalence, such that it can be directly applied to this query, to clearly illustrate the transformation for this query. The logical equivalence rule for the *IsIn* operator would actually be more general.

[L_2] `IsIn( Select(z, z in ext(Course) & z’ in RemEnrolled & PRED(t,z,z’)), y)`
          $\sim$ `y = z & z in ext(Course) & z’ in RemEnrolled & PRED(t,z,z’)`

After applying this equivalence, and simplifying to eliminate z (replacing with y) we obtain the following query, in which the expensive membership test *IsIn* has been eliminated:

```
Select(t.name1, t in RemStudent, y in ext(Course), &
          z' in RemEnrolled & PRED(t,y,z') & y.cname = ''cs101''
```

We then apply the equivalence H_2 for `ext(Course)`, and simplify further, to obtain the following query, which only performs join operations (and not a membership test):

```
Select(t.name1, t in RemStudent, & a in RemCourse & z' in RemEnrolled,
         & z'.name1 = t.name1 & z'.course1 = a.course1 & a.course1 = ''cs101''
```

4.2 An example applying semantic equivalences

This example demonstrates the application of semantic, logical and heterogeneous object equivalences, to produce a simplified equivalent query. Typically, semantic equivalences correspond to integrity constraints and in this example, we consider semantic equivalences in the global schema. Later, we discuss the issue of validating these equivalences in the remote databases. We consider a fairly complex query which is expressed using several membership tests, all of which are eliminated. We also demonstrate that simplification is possible, even with redundancy of the heterogeneous equivalences. In the global schema, both classes `Passenger` and `Flight` have attributes `departs` and `passengers`, respectively, which refer to each other. They are as follows:

Example 4.2

```
Global schema                Remote schema
Class  Passenger                Relation RemPassenger(Pno1, Pname1, Pcity1)
   { name: string
     departs: set(Flight)       Relation RemPassFlight(Pno1, Fno1)
     city: string
     Pno: integer }             Relation RemFlight(Fno1, Ftime1)

Class Flight
   { Fno: integer
     passengers: set(Passenger)
     time: float }
```

Consequently, the equivalence H_1 for `ext(Passenger)` refers to instances of `Flight` in the global schema, and to tuples in `RemPassenger`, `RemPassFlight`, etc., in the remote schema. The equivalence H_2 for `ext(Flight` is similar. They are as follows:

```
[H1] ext (Passenger) ~
            Select(NewObject([ name:=x.Pname1,
                               departs:= Select( y, y in ext(Flight), z in RemPassFlight,
                                                & z.Pno1 = x.Pno1 & z.Fno1 = y.Fno),
                              city:=x.Pcity1, Pno:= x.Pno1 ])
        x in RemPassenger )
∼ Select(NewObject([ name:=x.Pname1,
                     departs:= Select( y, y in ext(Flight), z in RemPassFlight,
                                       & PRED(x,y,z),
                     city:=x.Pcity1, Pno:= x.Pno1 ])
                     x in RemPassenger )
where PRED(x,y,z)  ∼ z.Pno1 = x.Pno1 & z.Fno1 = y.Fno
```

```
[H2] ext (Flight)  ∼
        Select(NewObject([ Fno:=t.Fno1, time:=t.Ftime1,
                           passengers:= Select( w, w in ext(Passenger),
                                                p in RemPassFlight,
                                                & p.Pno1 = w.Pno & p.Fno1 = t.Fno1) ]
                           t in RemFlight )
∼ Select(NewObject([ Fno:=t.Fno1, time:=t.Ftime1,
                     passengers:= Select( w, w in ext(Passenger), p in RemPassFlight,
                     & PRED(p,w,t)], t in RemFlight )
where PRED(p,w,t) ∼ p.Pno1 = w.Pno & p.Fno1 = t.Fno1
```

Finally, we have the following semantic equivalence S_1, in the global schema, expressing an integrity constraint between instances of Flight and Passenger, through attributes passengers and departs:

```
[S1] ∀ p in ext(Passenger) ∀ f in ext(Flight) IsIn(f.passengers, p)
      ∼ IsIn(p.departs, f)
```

The heterogeneous equivalences, combined with the semantic equivalence, provide multiple ways to obtain alternative queries, which are redundant. However, in many instances, there is only one correct simplified query and we demonstrate that the optimizer obtains this query. The following query selects the names of passengers, and co-passengers, from Paris on a flight that departs at 11:

```
Select([a.name, [c.name]], a in ext(Passenger), b in a.departs, c in b.passengers,
         & b.time = ''11h'' & a.city = ''Paris''
```

We first apply the semantic equivalence S_1, and we obtain 3 equivalent queries, as follows:

```
1. Select([a.name, [c.name]], b in ext(Flight), a in b.passengers, c in b.passengers,
           & b.time = ''11h'' & a.city = ''Paris''
```

```
2. Select([a.name, [c.name]], a in ext(Passenger), b in ext(Flight),
                       c in ext(Passenger),
             & b.time = ``11h'' & a.city = ``Paris''
             & IsIn(a.departs, b) &IsIn(d.passengers, c)
3. Select([a.name, [c.name]], a in ext(Passenger), c in ext(Passenger),
          & b in Intersection(a.departs, c.departs))
          & b.time = ``11h'' & a.city = ``Paris''
```

We choose to expand the first of these queries. However, we can demonstrate that all three queries will reduce to an identical query wrt the remote schema.

```
Select([a.name, [c.name]], a in ext(Passenger), b in a.departs, c in b.passengers,
         & b.time = ``11h'' & a.city = ``Paris''
```

We first apply heterogeneous equivalence H_1 to simplify the following query fragment:
`a.name { a in ext(Passenger) }` $\sim$

```
      Select(NewObject([name:=x.Pname1
                        departs:= Select(y, y in ext(Flight), z in RemPassFlight,
                               & PRED(x,y,z) ),
                        city:=x.Pcity1,
                        Pno:= x.Pno1 ]) x in RemPassenger ).name
```

$\sim$ `x.Pname1 {x in RemPassenger}`
We similarly simplify fragments `(c.name {c in ext(Passenger)})` , and `{b in ext(Flight) }` . The query fragment containing the *IsIn* membership operator can also be simplified. We first apply equivalence H_1 for `{a in ext(Passenger)}` and H_2 for `{b in ext(Flight)}` , and simplify, to obtain the following fragment (details are not shown here):
`IsIn(a.departs, b) { a in ext(Passenger), b in ext(Flight)}` $\sim$

```
       IsIn(Select(y, y in ext(Flight) & z in RemPassFlight, & x in RemPassenger
                                                              & PRED(x,y,z),
            NewObject([Fno:=t.Fno1,
                        time:=t.Ftime1,
                        passengers:= Select(w, w in ext(Passenger),
                              p in RemPassFlight, &PRED(p,w,t ]
                    t in RemFlight ) )
```

We can now apply a specific form of the logical equivalence, similar to L_2 which simplifies the *IsIn* operator. It was also used in the previous example. This equivalence allows us to substitute the second argument `NewObject`, for `y` in the first argument, for the *IsIn* operator. We omit the details here. After applying this equivalence, we simplify further, for those variables that occur in the arguments of PRED. This step eliminates the expensive membership operation, and produces the following equivalent query fragment:

`~ {p in RemPassFlight, x in RemPassenger, t in RemFlight & PRED(x,t,p)}`

We apply a similar simplification for the following fragment, to eliminate the membership test:

`IsIn(b.passengers, c) { b in ext(Flight), c in ext(Passenger) }`

We are also able to obtain the following simplifications for other fragments:

```
b.time { b in ext(Flight) }  ~ t.Ftime { t in RemFlight }
a.city { a in ext(Passenger) }  ~ x.Pcity1 { x in RemPassenger }
```

Finally, we obtain the following simplified query, in which `a`, `b` and `c` are substituted by `x`, `x'` and `t`, resp. Note that in the final query, we have completely eliminated the membership test. Also note that we obtain an identical query if we start with the other two equivalent queries as well. The final query is as follows: where there are no membership tests.

```
Select(([x.name, [x'.name] ], x in RemPassenger, x' in RemPassenger, t in RemFlight,
            p in RemPassFlight, w in RemPassFlight
            & t.time = ''11h'' & x.city = ''Paris''
            & PRED(x,t,p) & PRED(w,x',t)
```

```
where PRED(x,t,p) ~ P.Pno1 = x.Pno1 & z.Fno1 = t.Fno1
and PRED(w,x',t) ~ w.Pno1 = x'.Pno1 & w.Fno1 = t.Fno1.
```

4.3 Obtaining multiple equivalent queries

In this example, we apply semantic equivalences in the remote schema, and heterogeneous equivalences, to obtain multiple equivalent queries. The semantic equivalences correspond to implicit integrity constraints based on the uniqueness of a "key". The costs of each alternate query must be determined before choosing an optimal query. We also demonstrate that explicit joins may be replaced by path expressions, (which may be more efficient).

For example, in the global schema, relationships among object instances are explicit. Attributes `Eno` and `Pno` in the class `EmpProj` explicitly store the corresponding keys of `Employee` and `Project`, respectively. However, the remote schema in this example *is an object schema*, and there are object references in the remote schema, e.g., `manager1` of class `RemProject` refers to an instance of the class `RemEmployee`. Using a semantic equivalence in the remote schema, based on the implicit integrity constraint for keys, we are able to eliminate the joins in the simplified query. The schema are as follows:

Example 4.3

```
Global  schema                            Remote schema
Class  Project                                Class RemProject
   { Pno: integer                                { Pno1: integer
     Pmanager: integer }                           manager1: RemEmployee
                                                   workers1: set(RemEmployee) }
```

```
Class Employee
   { Eno: integer                          Class RemEmployee
     Ename: string                                { Eno1: integer
     Esal: float  }                                 salary1: float
                                                    worksin1: RemProject
Class EmpProj                                       Ename1: string }
   { Eno: integer
     Pno: integer }
```

The heterogeneous equivalences are as follows, with two equivalences H_3 and H_4 for `ext(EmpProj)`:

```
[H1] ext (Project) ~
        Select(NewObject([Pno:=x.Pno1, Pmanager:=x.manager1.Eno1],
        x in ext(RemProject) ))
[H2] ext (Employee) ~
        Select(NewObject( [Eno:=y.Eno1, Ename:=y.Ename1, Esal:=y.salary1],
                          y in ext(RemEmployee) ))
[H3] ext (EmpProj) ~ Select( NewObject( [Eno:= z2.Eno1, Pno:= z1.Pno1],
                            z1 in ext(RemProject), z2 in z1.workers1) )
~ [H4] ext(EmpProj) ~ Select(NewObject( [Eno := z.Eno1, Pno := z.worksin1.Pno1],
                            z in ext(RemEmployee) ))         [H4 is equiv. to H3]
```

We have the following semantics equivalences in the remote schema, wrt `RemEmployee` and `RemProject`, corresponding to an implicit integrity constraint, based on the uniqueness of the keys:
[S$_1$] ∀ `p1 in ext(RemProject)` ∀ `p2 in ext(RemProject) (p1 = p2)` ~ `(p1.Pno1 = p2.Pno1)`
[S$_2$] ∀ `e1 in ext(RemEmployee)` ∀ `e2 in ext(RemEmployee) (e1 = e2)` ~ `(e1.Eno1 = e2.Eno1)`

Consider a query in the global schema to retrieve project managers whose salary is less than that of their employees, as follows:

```
Select(x.Pmanager, x in ext(Project), y in ext(Employee), z in ext(EmpProj),
                                          t in ext(Employee)
        & x.Pno = z.Pno & y.Eno = z.Eno & x.Pmanager = t.Eno & t.Esal < y.Esal)

~

Select(x.Pmanager, x in ext(Project), y in ext(Employee), z in ext(EmpProj),
            t in ext(Employee) & PRED(x,y,z,t) & t.Esal < y.Esal)

where PRED(x,y,z,t)  ~ x.Pno = z.Pno & y.Eno = z.Eno & x.Pmanager = t.Eno
```

Note the explicit join, represented by `PRED(x, y, z, t)`, which is eventually eliminated by using path expressions in the simplified query, wrt the remote schema. We first apply heterogeneous equivalences H$_1$, H$_2$, and H$_4$, and perform some simplifications to obtain the following:

```
Select(a.manager1.Eno1, a in ext(RemProject), b in ext(RemEmployee),
                c in ext(RemEmployee), d in ext(RemEmployee),
          a.Pno1=c.worksin.Pno1 & b.Eno1=c.Eno1 & a.manager1.Eno1=d.Eno1
                &d.salary1 < b.salary1 )
```

We then apply the equivalence S_1 to the equality (`a.Pno1 = c.worksin.Pno1`), to obtain the substitution {`a` | `c.worksin` }. Similarly, from (`b.Eno1 = c.Eno1`) and S_2, we obtain the substitution {`b` | `c` }. Finally, we obtain the following simplified equivalent query, which has a *single explicit join* and several path expressions:

```
Select(c.worksin.manager.Eno, c in ext(RemEmployee), d in ext(RemEmployee),
           & c.worksin1.manager1.Eno1 = d.Eno1 & d.salary1 < c.salary1 )
```

Alternately, we can utilize the equivalences H_1, H_2 and H_3, perform some simplifications, and then apply S_1 and S_2 to obtain the following query, in which *all explicit joins* have been replaced by path expressions:

```
Select(c1.manager1.Eno1, c1 in ext(RemProject), c2 in c1.workers1
         & c1.manager1.salary1 < c2.salary1 )
```

The first query involves two variables that range over instances of the class `RemEmployee` and has one explicit join. The second query has one variable that ranges over the remote schema instances of `RemProject` and no explicit join. The cost for evaluating each of these queries would have to be estimated, before selecting the optimal query.

4.4 Simplifying a nested query and query decomposition

This example demonstrates the un-nesting of a query, together with the application of heterogeneous object equivalences, to obtain a query wrt the remote schema. The query may reference information in multiple remote schema. Thus, after obtaining a simplified query wrt the remote schema, the optimizer will use some syntactic analysis of the join conditions in the query, together with information about the co-location of relations in the remote databases, to obtain several possible (nested) sub-queries, one for each of the remote target databases.

In the example remote schema, we assume that `RemPeople` and `RemEarns` are in one remote database, and `RemReside` and `RemCity` are in another database. The schema are as follows:

Example 4.4

```
Global schema                           Remote schema #1

Class Person                               Relation RemPeople(name1, age1)
   {  name: string                         Relation RemEarns(name1, income1)
      age:  integer
      income: float }
```

```
                                          Remote schema #2
Class City
    {  cname: string                        Relation RemCity(cname1)
       residents: Set(Person) }             Relation RemReside(name1, city1)
```

We have the following heterogeneous equivalence rules:

```
H1 ext (City) ~
      Select(NewObject( [cname:= z1.cname1,
                        residents:=Select( y, y in ext(Person), & z2 in RemReside
                                                  & z2.name1 = y.name & z2.city1 = z1.city1) ],
                        z1 in RemCity ) )


H2 ext (Person) ~
      Select( NewObject([name:= z1.name1, age:=z1.age1, income:=z2.income1],
             z1 in RemPeople, z2 in RemEarns, & z1.name1 = z2.name1) )
```

The following nested query selects people who reside in the city Paris and have zero income:

```
Select(x.name, x in Select(y, z in ext(City) & y in z.residents, &z.cname=``Paris''
                                      y.income = 0), x.age > 60)
```

We first un-nest this to obtain the following query:

```
Select(y.name, z in ext(City) & y in z.residents & z.cname = ``Paris''
             & y.income = 0) & y.age > 60)
```

We then apply the equivalence H_2 for `ext(City)` and simplify to eliminate irrelevant variables, and obtain the following:

```
Select(y.name, z1 in RemCity
            & y in Select(p, p in ext(Person), & z2 in RemReside
                            & z2.name1 = p.name & z2.city1 = z1.city1)
            & z1.cname1 = ``Paris'' & y.income = 0 & y.age > 60 )
```

We now apply the equivalence H_1 for `ext(Person)` and simplify further, to obtain the following un-nested query:

```
Select(z3.name1, z1 in RemCity, z3 in RemPeople, z4 in RemEarns, z2 in RemReside
         & z3.name1 = z4.name1 & z2.name1 = z3.name1 & z2.city1 = z1.city1
         & z1.cname1 = ``Paris'' &z4.income1 = 0 & z4.age1 > 60 )
```

We stated earlier that relations `RemEarns` and `RemPeople` are co-located in remote database #1, and `RemCity` and `RemResides` are co-located in database #2. Based on a syntactic analysis of the join conditions of this query, we would obtain several possible (nested) sub-queries. We would need to evaluate the cost of each set of queries before determining an optimal plan. Some possibilities include (SQ_1, SQ_2) and (SQ_3, SQ_4, SQ_5), as follows:

```
$SQ_1$ Select( z3.name1, z4 in RemEarns, z3 in RemPeople
           & z4.income1 = 0 & z3.age1 > 60 & z3.name1 = z4.name1)
$SQ_2$ Select( z2.name1, z5 in $SQ_1$ & z1 in RemCity & z2 in RemReside
           & z2.name1 = z5.name1 & z2.city1 = z1.cname1 & z1.cname1 = ``Paris'')

$SQ_3$ Select( z3.name1, z4 in RemEarns, z3 in RemPeople
           & z4.income = 0 & z3.age > 60 & z3.name1 = z4.name1)
$SQ_4$ Select( z2.name1, z1 in RemCity, z2 in RemReside
           & z2.city1 = z1.cname1 & z1.cname1 = ``Paris'')
$SQ_5$ Select( x.name1, x in  $SQ_3$, y in $SQ_4$ & x.name1 = y.name1)
```

5 Summary and future research

In this paper, we defined the heterogeneous search space for a HDBMS, where the global schema and query language are based on the Flora language and object model, and where the target remote databases may be either relational or object-oriented. We have extended a modular rule-based query optimizer which inputs a query in the global schema and obtains equivalent queries (or multiple sub-queries) wrt the remote schema entities. The optimizer uniformly performs (1) syntactic simplification, (2) logical transformations based on logical equivalences wrt built-in operators in the global schema, (3) logical transformation based on semantic equivalences based on integrity constraints in either the global schema or the remote schema, and (4) heterogeneous simplification, based on heterogeneous object equivalences, which map remote schema entities to entities in the global schema. The Flora optimizer has been implemented to handle syntactic simplification, and logical transformations based on logical and semantic equivalences. It is being extended to perform heterogeneous equivalences as well.

There are a number of issues that require further research. First, the semantic equivalences are based on integrity constraints in the global schema or remote schema. The heterogeneous object equivalences are also based on implicit integrity constraints, depending on the notion of a unique key. The task of validating these equivalences in the HDBMS is critical, if we are to correctly transform queries wrt the remote databases. There are several possible strategies for validation. We may specify some queries, in the global schema, to validate integrity constraints, and transform these queries and generate a set of queries wrt the remote schema. The queries can then be validated against the target databases. A different approach is to extend the solver (corresponding to the algebraic language) to induce integrity constraints wrt each remote schema, for the constraints in the global schema. We are considering both of these approaches, in future research.

We have also not considered built-in operators in the schema of the target databases, and corresponding logical equivalences. We have made the assumption that new objects are always created in the global environment, when a query executes and imports data values from the target databases. However, we must consider alternative scenarios, where an object may already be materialized, or where some portion of the object may be materialized. We must also develop a cost model for the HDBMS. We also have to address the problem of determining a search strategy, to enumerate the execution plans so that the best one may be selected.

Finally, in this paper, we assumed a global schema architecture for the HDBMS. We can extend our results to define a heterogeneous search space for other architectures, e.g., a federated HDBMS or a canonical HDBMS architecture.

6 Bibliography

Abiteboul, S. and Bonner, A. (1991) "Object and views," *Proc. of the SIGMOD International Conference*, 238-247.

Ahmed, R., de Smedt, P., Du, W., Kent, W., Ketabchi, M.A., Litwin, W.A., Rafii, A. and Shan, M.-C. (1991) "The Pegasus heterogeneous multidatabase system," *IEEE Computer, 24(12)*, 19–27.

Albert, J., Ahmed, R., Ketabchi, M., Kent, W. and Shan, M.-C. (1993) "Automatic importation of relational schemas in Pegasus," *Proc. of the Workshop on Research Issues in Data Engineering, ICDE-93.*

Arens, Y. and Knoblock, C.A. (1992) "Planning and reformulating queries for semantically-modeled multidatabase systems," *Proceedings of the First International Conference on Knowledge Management.*

Arens, Y., Chee, C.Y., Hsu, C.-N., Knoblock, C.A. (1993) "Retrieving and integrating data from multiple information sources," *International Journal of Intelligent and Cooperative Information Systems. Vol. 2, No. 2.*, 127-158.

Bancilhon, F., Cluet, S. and Delobel, C. (1989) "Query languages for object-oriented database systems," *Proceedings of the International Conference on Database Programming Languages.*

Barsalou, T. and D. Gangopadhay (1992) "M(DM): An open framework for interoperation of multimodel multi-database systems," *Proceedings of the International Conference on Data Engineering.*

Batini, C., Lenzerini, M. and Navathe, S.B. (December 1986) "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys, Vol. 18, No. 4*, 323-364.

Chang, Y., Raschid, L. and Dorr, B.J. (1994) "Transforming queries from a relational schema to an equivalent object schema: a prototype based on F-logic," *Proc. of the ISMIS-94 International Symposium.*

Chomicki, J. and Litwin, W. (1994) "Declarative definition of object-oriented multidatabase mappings," in *Distributed Object Management*, Oszu, M.T., Dayal, U. and Valduriez, P. (eds.) Morgan Kauffman Publishers.

Dayal, U. and Hwang, H. (1984) "View definition and generalization for database integration in a multidatabase system," *IEEE Transactions on Software Engineering, 10(6)*, 628-645.

Du. W., Krishnamurthy, R. and Shan, M.-C. (1992) "Query optimization in heterogeneous DBMS," *Proceedings of the International Conference on Very Large Data Bases.*

Florescu, D. and Valduriez, P. (1994) "Rule-based query processing in the IDEA system," *Proc. of the Intl. Symp. on Advanced Database Technology and their Integration*, Nara, Japan, October 1994.

Guting, R.H. (1993) "Second-order signature: a tool for specifying data models, query processing and optimization," *Proceedings of the International Conference on the Management of Data.*

Kent, W. (1991) "Solving domain mismatch and schema mismatch problems with an object-oriented database programming language," *Proceedings of the International Conference on Very Large Data Bases.*

Kifer, M., Kim, W. and Sagiv, Y. (1992) "Querying object-oriented databases," *Proc. of the ACM Sigmod Conference.*

Kim, W., Choi, I., Gala, S. annd Scheevel, M. (1993) "On resolving schematic heterogeneity in multidatabase systems," *Distributed and Parallel Databases, 1(3)*, 251–279.

Kim, W. and Seo, J. (December, 1991) "Classifying schematic and data heterogeneity in multidatabase systems," *IEEE Computer*, pages 12–18.

Krishnamurthy, R., Litwin, W. and Kent, W. (1991) "Language features for interoperability of databases with schematic discrepancies," *Proceedings of the ACM Sigmod Conference.*

Lakshmanan, L.V.S., Sadri, F. and Subramanian, I.N. (1993) "On the logical foundations of schema integration and evolution in heterogeneous database systems," *Proc. of the DOOD Conf.*, 81-100.

Novak,M., Gardarin, G. and Valduriez, P. (1994) "Flora: a functional-style language for object and relational algebra," , INRIA Tech. report.

Qian, X. (1993) "Semantic interoperation via intelligent mediation," *Proc. of Workshop on Res. Issues in Data Engg.*.

Qian, X. and Raschid, L. (1995) "Query interoperation among object-oriented and relational databases," *Proceedings of the International Conference on Data Engineering.*

Raschid, L., Chang, Y. and B. Dorr (1994) "Query Transformation Techniques for Interoperable Query Processing in Cooperative Information Systems," *Proceedings of the CoopIS International Conference.*

Raschid, L. and Chang, Y. (1994) "Interoperable query processing from object to relational schemas based on a parameterized canonical representation," *Submitted to the IJICIS journal.*

Sheth, A. and Larson, J. (1990) "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys, 22(3)*, 183–236.

Appendix A: Flora Types

Definition 1 *$(\mathcal{B}, \sum_{\mathcal{B}})$ is a built-in signature , where $\mathcal{B}$ is the set of Flora built-in types and $\sum_{\mathcal{B}}$ is the set of Flora built-in operators, with signatures. Currently the Flora built-in types $\mathcal{B}$=\{* **int**, **bool**, **float**, **string**\}. *At present, $\sum_{\mathcal{B}}$ contains the following operators:*

*+, -, *, / :* $\mathbf{float}\times\mathbf{float}\rightarrow\mathbf{float}$	*and, or :* $\mathbf{bool}\times\mathbf{bool}\rightarrow\mathbf{bool}$
not : $\mathbf{bool}\rightarrow\mathbf{bool}$	$<, \leq, >, \geq$ *:* $\mathbf{float}\times\mathbf{float}\rightarrow\mathbf{bool}$
contains, like : $\mathbf{string}\times\mathbf{string}\rightarrow\mathbf{bool}$	*=:* $s \times s \rightarrow\mathbf{bool}$ *for all type s*
union, difference, intersection : $\{s\} \times \{s\} \rightarrow \{s\}$ *for all type s*	
card : $\{s\} \rightarrow\mathbf{float}$ *for all type s*	*IsIn :* $\{s\} \times s \rightarrow\mathbf{bool}$ *for all type s*
IsEmpty : $\{s\}\rightarrow\mathbf{bool}$ *for all type s*	*max, min, avg :* $\{\mathbf{float}\}\rightarrow\mathbf{float}$.

Definition 2 *The algebra specific to an application domain is described by an object signature $(\mathcal{O}, \sum_{\mathcal{O}}, \prec, \mathcal{I})$, where $\mathcal{O}$ is the set of object type names, $\sum_{\mathcal{O}}$ is the set of user-defined operators (with their signatures), $\prec$ is a partial order on $\mathcal{O}$ (corresponding to the inheritance relationship), and I is a function which associates a state with every object type name $o \in \mathcal{O}$.*

Definition 3 *The set $\mathcal{S}_{\mathcal{B},\mathcal{O}}$ of Flora type expressions is an infinite set, recursively defined as follows:*
(i) if $b \in \mathcal{B}$ then $b \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$; *(ii) if $o \in \mathcal{O}$ then $o \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$;*
(iii)(set type constructor) if $s \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$ then $\{s\} \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$;
(iv)(tuple type constructor) if $s_1, \cdots, s_n \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$ and $f_1, \cdots, f_n$ are attribute names, then $[f_1 : s_1, \cdots, f_n : s_n] \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$ for every $n \geq 1$.

Definition 4 *NV is a particular set of variables, the named variables, representing the persistent roots of the databases. Each named variable x_i is associated with a name and a type, $type(x_i) \in \mathcal{S}_{\mathcal{B},\mathcal{O}}$. For example, object type extensions represent particular named variables in an object schema.*

Appendix B: Flora Expressions

Definition 5 *The set $T(X)$ of well-typed terms, using variables X, is inductively constructed.*
(i) If c is a constant of type s, then $c\in T(X)$ and it is of type s.
(ii) If $x\in X$ is a variable of type s, then $x\in T(X)$ and it is of type s.
*(iii)(**Operator application:**) If op is a built-in or user-defined operator, with signature $op{:}s_1 \times \cdots \times s_n \rightarrow s$, where $t_1, \cdots, t_n \in T(X)$ are of type $s_1, \cdots, s_n$, resp., then $op(t_1,\cdots,t_n)\in T(X)$, and it is of type s.*
*(iv)(**Tuple constructor:**) If $t_1, \cdots, t_n \in T(X)$, and they are of type $s_1, \cdots, s_n$, resp., and $f_1, \cdots, f_n$ are attribute names, then $[f_1 := t_1, \cdots, f_n := t_n] \in T(X)$, and it is of type $[f_1 : s_1, \cdots, f_n : s_n]$ $(n\geq 1)$.*
*(v)(**Attribute selector:**) If $t\in T(X)$ and it is of type $[f_1 : s_1, \cdots, f_n : s_n]$, then $t.f_i \in T(X)$, and it is of type s_i $(1\leq i\leq n)$.*
*(vi)(**Set constructor:**) If $t_1,\cdots,t_n \in T(X)$, and they are of type s, then $\{t_1,\cdots,t_n\} \in T(X)$ and it is of type $\{s\}$ $(n\geq 0)$.*
*(vii)(**Object dereferencor:**) If $t\in T(X)$ is of type $o \in \mathcal{O}$, then $t.state \in T(X)$ is of type $I(o)$.*
*(viii)(**Object constructor:**) If $o \in \mathcal{O}$ is an object type name, and $t\in T(X)$ is of type $I(o)$, then* **new** *$o(t) \in T(X)$ and it is of type o.*
*(ix)(**Forall, Exists operators:**) If $C\in T(X)$ is of type $\{s\}$, and pred$\in T(X \bigcup \{x\})$ is of type* **bool**, *then forall $(x$ in C, pred$)$ and exists$(x$ in C, pred$)\in T(X)$ are of type* **bool**.
*(x)(**Selection:**) If $C_1 \in T(X)$ is of type $\{s_1\}$, $C_2 \in T(X \bigcup \{x_1\})$ is of type $\{s_2\}$, $\cdots$, $C_n \in T(X \bigcup \{x_1, \cdots, x_{n-1}\})$ is of type $\{s_n\}$, pred$\in T(X \bigcup \{x_1, \cdots, x_n\})$ is of type* **bool**, *proj$\in T(X \bigcup \{x_1, \cdots, x_n\})$ is of type s. Assume that $\{x_1, \cdots, x_n\}$ and X are disjoint. Then,*
select *(proj, x_1 in C_1 and x_2 in C_2 and $\cdots$ and x_n in C_n, pred$)$ $\in T(X)$ and it is of type $\{s\}$ $(n\geq 1)$.*