Bulk Loading Techniques for Object Databases and an Application to Relational Data

Sihem Amer-Yahia, Sophie Cluet and Claude Delobel*

contact author: Sihem Amer-Yahia, INRIA, BP 105, 78153 Le Chesnay, France
tel: (33) 1 39 63 57 52, fax: (33) 1 39 63 56 74, email: Sihem.Amer-Yahia@inria.fr
author in the program committee: Sophie Cluet, INRIA, BP 105, 78153 Le Chesnay, France tel: (33) 1 39 63 58 20, fax: (33) 1 39 63 56 74, email: Sophie.Cluet@inria.fr

Abstract

We present a framework for designing, in a declarative and flexible way, efficient migration programs and an undergoing implementation of a migration tool called RelOO whose targets are any ODBC compliant system on the relational side and the O_2 system on the object side. The framework consists of (i) a declarative language to specify the migration process and (ii) an algebra-based program rewriting technique. The main originality of the language resides in the fact that it does not only provide the means to specify the database transformations from relations to objects, but also physical properties we want to achieve on the object side (such as clustering and sorting). The originality of the rewriting technique relies in its ability to optimize the migration processing time while taking into account physical properties and transaction decomposition. To achieve this, we introduce equivalences of a new kind that consider side-effects on the object database.

1 Introduction

The efficient migration of bulk data into object databases is a problem frequently encountered in practice. This occurs when applications are moved from relational to object systems, but more often, in applications relying on external data that has to be refreshed regularly. Indeed, one finds more and more object replications of data that are used by satellite Java or C++ persistent applications. For example, we know that the O_2 database system from O_2Technology [39] is used in that manner on industrial databases supporting several gigas of data.

The migration problem turns out to be extremely complicated. It is not rare to find migration programs requiring hours and even days to be processed. Furthermore, efficiency is not the only aspect of the problem. As we will see, flexibility in terms of database physical organization and decomposition of the migration process is at least as important. In this paper, we propose a solution to the data migration problem providing both flexibility and efficiency. The main originality of this work is an optimization technique that covers the relational-object case but is general enough to support other migration problems such as object-object, ASCII-object or ASCII-relational. In order to validate our approach, we focus on the relational to object case.

Relational and object database vendors currently provide a large set of connectivity tools and some of them have load facilities (e.g., ObjectStore DB Connect [27], Oracle V8 [28],

*University of Paris XI, LRI, 91405 Orsay Cedex, France.

O2DBAccess [26], GemConnect [17], ROBIN [14], Persistence [20], OpenODB [30]). However, these tools are either not flexible, or inefficient or too low-level. So far, the scientific community has mainly focused on modeling issues and has provided sound solutions to the problem of automatically generating object schemas from relational ones (e.g. [23, 15, 22, 18, 25, 21, 32, 12, 33, 16, 8, 19]). Our approach is complementary and can be used as an efficient support for these previous proposals. In a manner similar to [42, 40, 41], we focus on the problem of incremental loading of large amount of data in an object database taking into account transaction decomposition. Additionally, we propose optimization techniques and the means to support appropriate physical organization of the object database.

One major difficulty when designing a tool is to come up with the appropriate practical requirements. Obviously, every tool should provide efficiency and user-friendliness. In order to understand what more was required from a relational-object migration system, we interviewed people from O_2Technology. They gave us some insights on what the users of their technology were asking for. This led us to the conclusion that flexibility was the most important aspect of the problem. By flexibility we mean:

- Users should have the possibility to organize the output object database according to the needs of applications that will be supported on it. For instance, they should be able to specify some particular clustering of objects on disk.
- Users should be able to slice the migration process into pieces that are run whenever and for as long as appropriate so that (i) relational and object systems are not blocked for long periods of time or at a time that is inappropriate (i.e., when there are other important jobs to perform) and (ii) a crash during the migration process entails minimal recovery.

In this paper, we present (i) a framework for designing, in a declarative and flexible way, efficient migration programs and (ii) an undergoing implementation of a migration tool called RelOO. The framework consists of:

- A declarative language to specify the migration process. The main originality of the language resides in the fact that it does not only provide the means to specify database transformations from relations to objects, but also physical properties we want to achieve on the object side (such as clustering and sorting). This language is simple but covers most practical needs. Notably, it allows to capture most automatically generated mappings as proposed by, e.g., [23, 15, 22, 18, 25, 21, 32, 12, 33, 16, 8, 19].
- An algebra-based rewriting technique whose originality lies in its ability to optimize the migration processing time while taking into account physical properties and transaction decomposition. To achieve this, we introduce equivalences of a new kind that consider side-effects on the object database. This constitutes the main contribution of the paper.

  An important characteristic of the optimization technique is that it is not restricted to the "relations towards objects" case. It can be used to support dumping and/or reloading of any database or other kinds of migration processes (objects to objects, ASCII to objects, etc.).

The RelOO system, we present here, is an implementation of this general framework with a given search strategy. Its targets are any ODBC compliant system on the relational side and the O_2 system on the object side.

In this paper, we assume that there are no updates to the relational database during the migration process. This is a strong limitation but we believe this is an orthogonal issue that does not invalidate our approach. The declarativity of the migration specification language we propose should allow the use of the systems' log mechanism to take updates into account.

The paper is organized as follows. Section 2 explains the migration process. Section 3 introduces our language. In Section 4, we present the algebraic framework, investigate various migration parameters and give some examples of appropriate program rewriting. In Section 5, we give a short overview of the RelOO system optimizer and code generator. Section 6 concludes this paper.

2 The Migration Process

Figure 1 illustrates our view of the migration process. The user specifies the process by means of a program containing two parts that concern (1) the logical transformations from relational to object schemas and bases, and (2) the physical constraints that should be achieved.

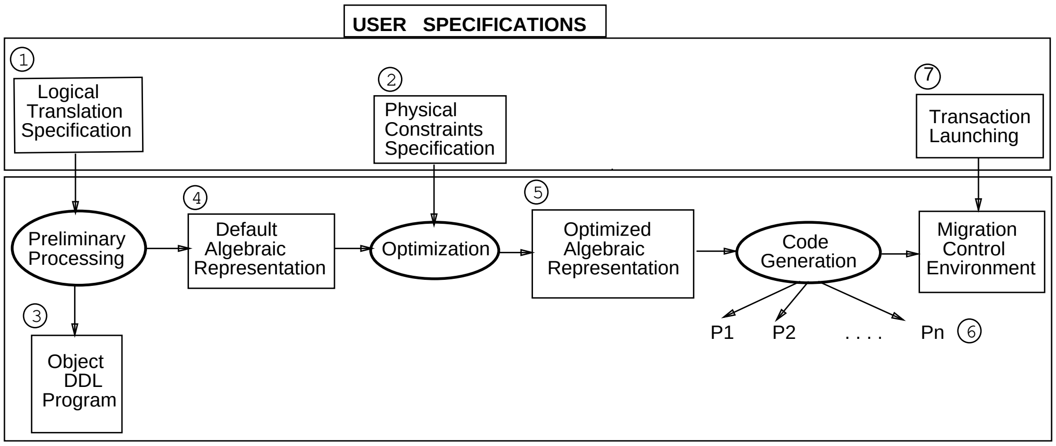

Figure 1: *The Migration Process*

The translation specification is given to a first module that generates (3) the object schema and (4) a default algebraic representation of the migration process. (5) Optimization based on rewriting follows taking into account user physical constraints. (6) Then, migration programs are generated that can be parameterized by load size or processing time. (7) To achieve the migration, the user (or a batch program) launches these programs with the appropriate parameters whenever the system is available (i.e., not busy with other important processing).

In this paper, we describe the RelOO language to specify logical translations and physical constraints, the rewriting technique and the optimizer and code generator we implemented. We do not describe the object schema construction which is rather obvious.

3 The Specification Language

The RelOO language is simple and declarative. It is used to express logical database transformations as well as physical database organization. The complete syntax of this language is given in [4].

3.1 Logical Database Transformation

Database schema transformations from relational to object systems have been largely addressed. Notably, semantical relational properties such as normal forms, keys, foreign keys, existing relationships and inclusion dependencies have been used to automatically generate a *faithful* object image of the relational schema. We do not address this issue here but provide a simple language that can be used to implement these approaches and that supports efficient optimization strategies.

Our database transformation language is defined in the same spirit as existing object views languages (e.g. [1, 35, 9, 34, 13]). Although simple, it can be used to specify most practical database transformations. Classes (generated from one or several relations), simple and complex attributes, class hierarchies, aggregation links and ODMG–like relationships [6] can be created from any relational database. Because we focus on data migration, we illustrate only some of the language features using a simple example. Part of this example will be used in the sequel to discuss the efficiency and flexibility issues that constitute our core contribution. Consider the following relational schema:

r_employee[emp:integer,lname:string,fname:string,town:string,function:string,salary:real]
r_project[name:string,topic:string,address:string,boss:integer]
r_emp_prj[project:string,employee:integer]

Relations *r_employee* and *r_project* provide information on, respectively, employees and projects in a research institute. In our example, they are used to construct three object classes (`Employee`, `Researcher` and `Project`). The underlined attributes are the keys of each relation. These are required to maintain existing links between tuples in the relational database and their corresponding created objects. This is needed to be able to maintain the consistency between relational and object databases (problem that we do not address here) but also to compute cyclic references in the object base. Relation *r_emp_prj* captures the many-many relationship existing between employees and projects: a project contains several employees and an employee may belong to several projects.

Let us now consider an example of database transformations as given in the following RelOO translation specification.

```
create class     Employee
from relations   r_employee
with attributes  name:string is lname+fname,
                 salary:real is salary
extent           Employees
where            function ≠ 'Researcher'
                 and town = 'Rocquencourt'

create class     Researcher
from relations   r_employee
inherits         Employee
extent           Researchers
where            function = 'Researcher'
                 and town = 'Rocquencourt'

create class     Project
from relations   r_project
with attributes  name:string is name,
extent           Projects
where            address = 'Rocquencourt'

create link      Project,Employee
from relations   r_employee, r_project
with attributes  manages:Project
                 in class Employee
where            boss = emp

create link      Project,Researcher
from relations   r_emp_prj, r_employee, r_project
with attributes  staff:set(Researcher) in class Project,
                 projects:set(Project) in class Researcher
where            employee = emp and project = name
```

The `create class` statements create classes which attributes are derived from relational ones by applying simple functions (identity, string concatenation, arithmetic functions,...). Note that the relation *r_employee* has been partitioned to form two classes (`Employee` and `Researcher`) and that only employees and projects located in "Rocquencourt" will be migrated. This is formulated in the `where` clauses. Class `Researcher` inherits from class `Employee`. This means that the former inherits the attribute specification of the latter (since there is no redefinition). However, their population are specified in a disjoint manner through their respective `where` clauses. Finally, note that we give a name to class extents. This is required in order to support persistence by reachability. Unless specified otherwise, the extent of a class does not contain the objects of its subclasses. For instance, the name `Employees` will be related to all employees who do not live in "Rocquencourt" and are not researchers ($function \neq' Researcher'$).

The two `create link` statements define, respectively, a reference link `manages` and an ODMG relationship (attribute and its inverse) `staff/projects`. The first one links each employee in Class `Employee` to the projects he/she manages (if he/she is a manager). Since, researchers are employees, they inherit this link. The second one relates projects and researchers and implies a filtering of the relation *r_emp_prj* so as to reject employees who are not researchers and projects which are not located in "Rocquencourt". This will be illustrated in the sequel. A `create link` statement contains three clauses that specify, respectively, the relations involved, the object attributes that materialize the link in each class and the join condition on the given relations. Note that there is no condition on the existence of foreign keys and inclusion dependencies. The user can construct any link he wants by giving an appropriate condition in the `where` clause. In the example, the aggregated attributes are of type `set`. Other collection types (e.g. `list`, `bag`) can also be used.

When the translation specification is processed, a schema definition is generated and compiled in the target object system. We give below the class definitions generated by the prototype we implemented on top of O_2. Relationships are materialized by two apparently disconnected attributes that enable the use of logical connections between the related objects. Creation and update methods guarantee the referential integrity of the relationship. However, relying on these methods at migration time is not always a good solution in terms of efficiency since (i) method calls are expensive and (ii) potentially involve random accesses to the store. We will see that our optimization technique proposes various ways to process relationships.

```
class Employee type
tuple(name:string,
salary:real,
manages:Project)

class Researcher inherits Employee
public type tuple(
    projects:set(Project))

class Project type
tuple(name:string,
staff:set(Researcher))

name Employees:set(Employee);
name Researchers:set(Researcher);
name Projects:set(Project);
```

3.2 Physical Database Organization

The object database could be organized physically after migration time. However, this would require very long processing. It is thus vital to integrate physical requirements in the actual migration. We consider two physical properties: *clustering and ordering.*

Clustering techniques are provided by most object database systems with different approaches and have been studied intensively (e.g., [7, 37]). The goal of a cluster is to minimize page faults by placing objects that are often accessed together as close as possible on disk. In RelOO, we do not assume any particular system clustering policy. We provide a primitive to specify clusters and a rewriting mechanism that takes this specification into account. The main idea is that the object system is in a better position to efficiently create a cluster if it is given all cluster-related objects at the same time. This is indeed true for our target system O_2 but also for, e.g., ObjectStore.

The ODMG model (and most object systems) features list and array constructors that are used to access objects according to some logical order. A list (or array) of objects is usually represented on disk as a list of object identifiers. If the order of objects on disk is different from the list logical order, one risks random and expensive access to the secondary storage. Again, we do not assume any particular system implementation but rely on the fact that it is easier to place objects on disk in a given order if the migration input respects this order. As a matter of fact, our implementation on top of the O_2 system guarantees a correct ordering.

Below are two RelOO statements. One specifies that only "Network" projects should be clustered with their `staff` attribute (not the related researcher objects, just the `staff` set of object identifiers). Deeper clusters can be specified. It is the case for the other cluster statement in which projects should be ordered by name (the order is by default ascendant) and each `Project` object should be clustered in a file with its `Researcher` objects (via the `staff` attribute).

`class`	Project	`class`	Project
`if`	topic='Network'	`order by`	name
`cluster on`	staff	`cluster on`	staff(Researcher)

Object sharing makes it impossible to guarantee that all database researchers will be clustered with their projects. A cluster is just an "indication" of what should be achieved. However, a good cluster specification may seriously improve the performance of object applications. The order in which cluster specifications are given is important.

Two specifications may be contradictory. This is obviously the case for the two above ones. Other contradictions are less easy to detect: e.g., specifying an order on researchers independently from the projects to which they belong, may be inconsistent with the `Project` cluster on `staff(Researcher)` since it entails that researchers are created according to their membership to a project. The RelOO compiler checks the consistency of the given specifications using techniques similar to those found in [7].

4 Rewriting Migration Programs

Given a translation specification, a first module generates an object schema and a default algebraic expression representing the migration process. This expression is given to the optimization module which uses physical properties for further processing.

The optimization process relies on rewriting based on algebraic equivalences. The rewriting is performed according to a search strategy and a cost model, both of which will be discussed in Section 5. In the current section, we present the algebra and the rewriting possibilities it offers.

Algebras are commonly used to optimize queries. However, queries do not have side-effects whereas we obviously are in a context with many of them (since the goal is the creation of an object database). Our choice of an algebraic tool can thus be argued. It was motivated by the following facts: (1) the side-effects operations are not arbitrary and are always applied on the result of queries; (2) the potential for optimization relies on the rewriting of these queries. Thus, we propose a representation of the migration program mainly composed of standard algebraic operations. An operator named Map is added to apply database update operations on each element of its input set.

We now present the default program representation, explain how side-effects can be taken into account by the rewriting process, investigate the parameters that should be considered by the optimizer and illustrate the rewriting possibilities using some examples.

4.1 Default Algebraic Representation

The algebra we use is that of [11]. It is well adapted to the problem at hand since it manipulates sets of tuples, the attributes of which can be atoms, objects or complex values. Thus, it captures both object and relational models. The algebraic operators are, for the most part, standard. The main difference with the relational algebra is that attributes may be complex, thus sometimes involving the use of nested algebraic expressions. Note that this algebra also fits nested relations or complex objects. It can also be used on ASCII data files as long as there exists a mapping between this data and some database representation (relational or other). Thus, the framework we propose can indeed cover most data migration cases.

The default representation corresponds to a "naive"[1] evaluation of data migration. It consists of a set of algebraic expressions (also called **blocks**) that correspond to the translation program statements. Some of the blocks create objects, others evaluate complex attributes and relationships. The generated blocks can be executed in any order. As we will see, this is possible because of the special object creation primitive we use that (i) keeps trace of the created objects and (ii) guarantees that we do not create two objects corresponding to the same relational key. However, to achieve rewriting, a default order (the one in which the translation statements are given) is chosen. Let us consider now a subpart of the translation program given in Section 3 that creates two classes (`Researcher` and `Project`) and one relationship `staff/projects`. Figure 2 shows two blocks $\mathbf{B}_{\mathrm{Res}}$ and $\mathbf{B}_{\mathrm{Prj}}$ that create, respectively, the `Researcher` and `Project` objects.

Both blocks are similar. The upper part of $\mathbf{B}_{\mathrm{Res}}$ features a selection operation corresponding to the `where` clause of class `Researcher` definition. Next, comes a projection on the relational attributes that are involved in the transformation. Then, a Map operation creates one object for each input tuple and memorizes the newly created objects (attribute o). This Map uses the obj function whose input is a relational key. In our example, this key is composed of a single attribute emp, more can be used. The obj function has two effects:

1. It maintains the correspondence between relational and object entities in a table. There is one table per couple class/relation, each containing relational keys and object identifiers. In our implementation, these tables are indexed on relational keys. Correspondence tables can grow considerably and require good storage and access policies as found in [40]. We will see that we may avoid maintaining them in some cases by applying some rewriting techniques.

2. It returns an object identifier that is either created (when a key value is first encountered) or retrieved from the correspondence table (we will see some examples of this later on).

Finally, two Map operations are used to update the attributes `name` and `salary` of the `Researcher`

[1] We will see in the sequel that a naive algorithm can be appropriate in some cases.

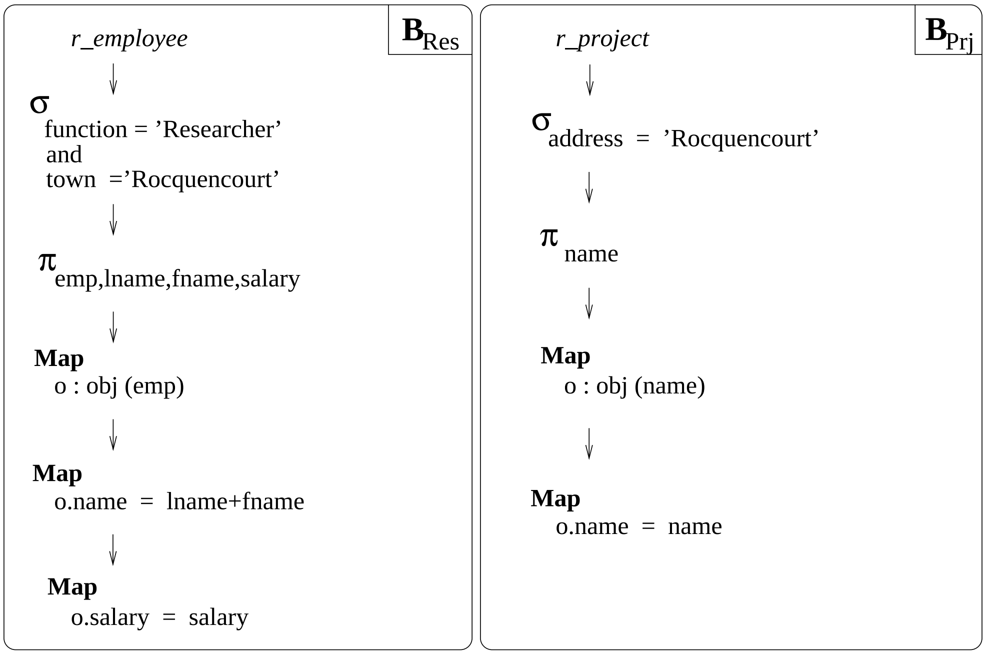

Figure 2: *Researchers and Projects*

objects [2].

Let us now consider a relationship specification: e.g. the `staff/projects` relationship. For each direction of a relationship, a block is created (see Figure 3).

Both $\mathbf{B}_{\text{staff}}$ and $\mathbf{B}_{\text{projects}}$ offer the same pattern. Let us consider $\mathbf{B}_{\text{staff}}$. Two join operations filter the *r_emp_prj* relation. This avoids creating relationships between objects that should not be in the database (i.e., projects or employees that are not part of the "Rocquencourt" site) or that should not be related (i.e., projects and non-researcher employees). A projection follows to remove irrelevant attributes. Then, researchers are grouped according to their membership to a project. This is done using the *Group* operator. This operator has three parameters: the name of the attribute that will denote the grouped elements (g), the name(s) of the grouping attribute ($name$) and the attribute(s) on which each group will be projected (emp). Therefore, a set g of related emp values is associated to each distinct $name$ value. The last operation is a Map that assigns to the `staff` attribute of each `Project` object ($obj(name)$ retrieves the appropriate object identifier from the correspondence table `(Project,r_project)`) its value (a set of `Researcher` objects). This value is computed by applying the $obj(emp)$ operation to each element of the set g (embedded Map operation).

4.2 Taking Side-Effects Into Account

In this section, we introduce some concepts that will allow, among other things, to consider optimization of the migration beyond block boundaries. So far, we have presented the default migration representation as a set of distinct algebraic blocks. In that context, it is difficult to detect optimization that would spread across two blocks, e.g., that would replace the two blocks $\mathbf{B}_{\text{Prj}}$ and $\mathbf{B}_{\text{staff}}$ by a single one both creating projects and assigning their `staff` attribute. To do that, we need to provide explicitly semantic connections between the various blocks.

[2] For an easier presentation, we simplified the way object creation and updates are performed. For instance, object creation requires the pair class/relation involved (i.e, we write $obj(emp)$ instead of $obj_{c,r}(emp)$)

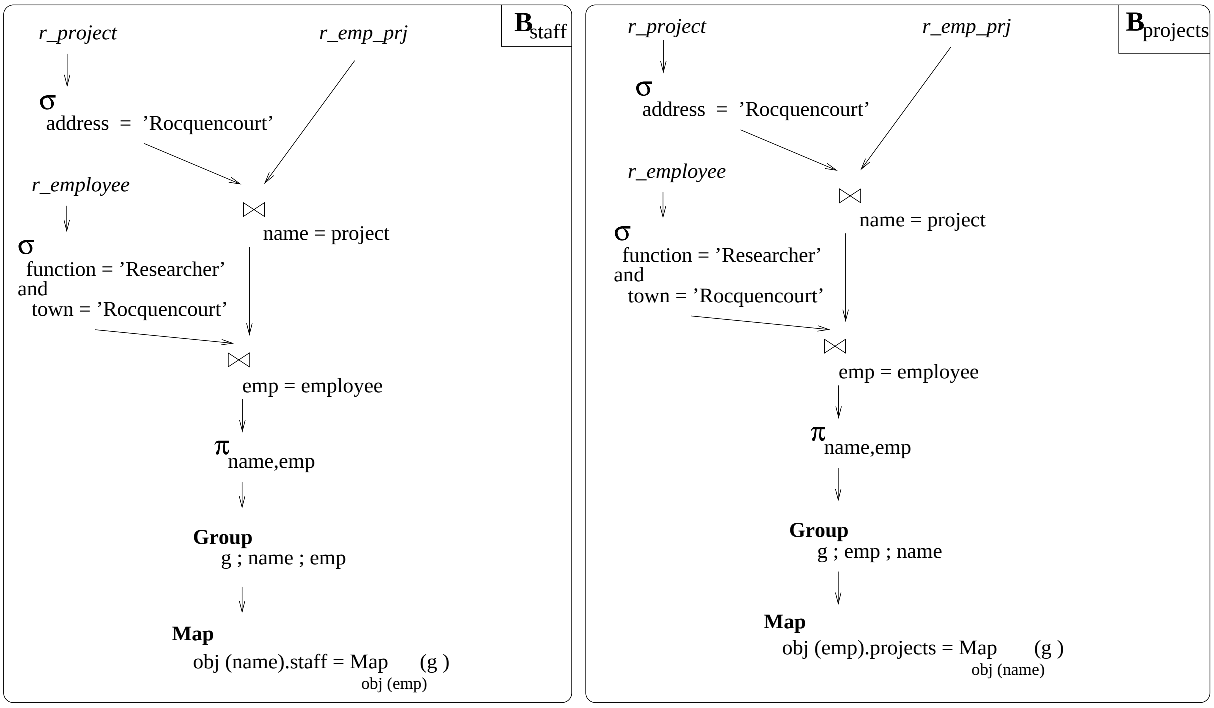

Figure 3: *The staff/projects Relationship*

Obviously, the connection we are interested in is not the standard one through algebraic data flow i.e., the input of one operation is the output of the previous one. There are two reasons for that: (i) we may want to discard previous results (e.g., we do not need the researchers data flow in order to create the projects) and, more importantly, (ii) rather than proving equivalences involving the data flow, we are interested in maintaining the logical database state (i.e, the object database resulting from a sequence of operations).

Thus, during the optimization process, an essential parameter is the database state that links together both algebraic operations and blocks. We call this parameter DS (for Database State). Now, each algebraic operation is considered with respect to the standard data flow and the DS parameter (see [5] for more details on the effect of each algebraic operation on DS). This is detailed for block $\mathbf{B}_{\mathrm{Res}}$ on the left part of Figure 4 which shows how the DS parameter is altered by each operation. Note that only Map operations have side effects and thus modify DS. DS is represented by a table where an entry is defined for a pair class/source relation. An entry in the table has two fields: *pred* and *atts*. The former is a predicate representing the selected tuples in the relation that are used to create the objects in the class. The latter represents the objects attributes that have been assigned along with their computed value. An empty DS represented by the symbol $\emptyset$ (the object database has not been populated yet) is assigned to block $\mathbf{B}_{\mathrm{Res}}$ and is not modified by the three first operations. After the $Map_{o:obj(emp)}$ operation, DS has a new entry whose *pred* field indicates the objects that have been created (one per tuple of the *r_employee* relation satisfying the predicate). The second and third Map operations then update the *atts* field.

The DS of a block is the DS associated with the last operation in the block. At the termination of a block, the standard algebraic data flow is discarded while DS is passed to the next block that will modify it further[3]. This is illustrated in the right part of Figure 4 which shows the DS obtained after each block of the default algebraic representation.

[3]On figures, we use a thin line to represent the standard data flow and a dash line for DS

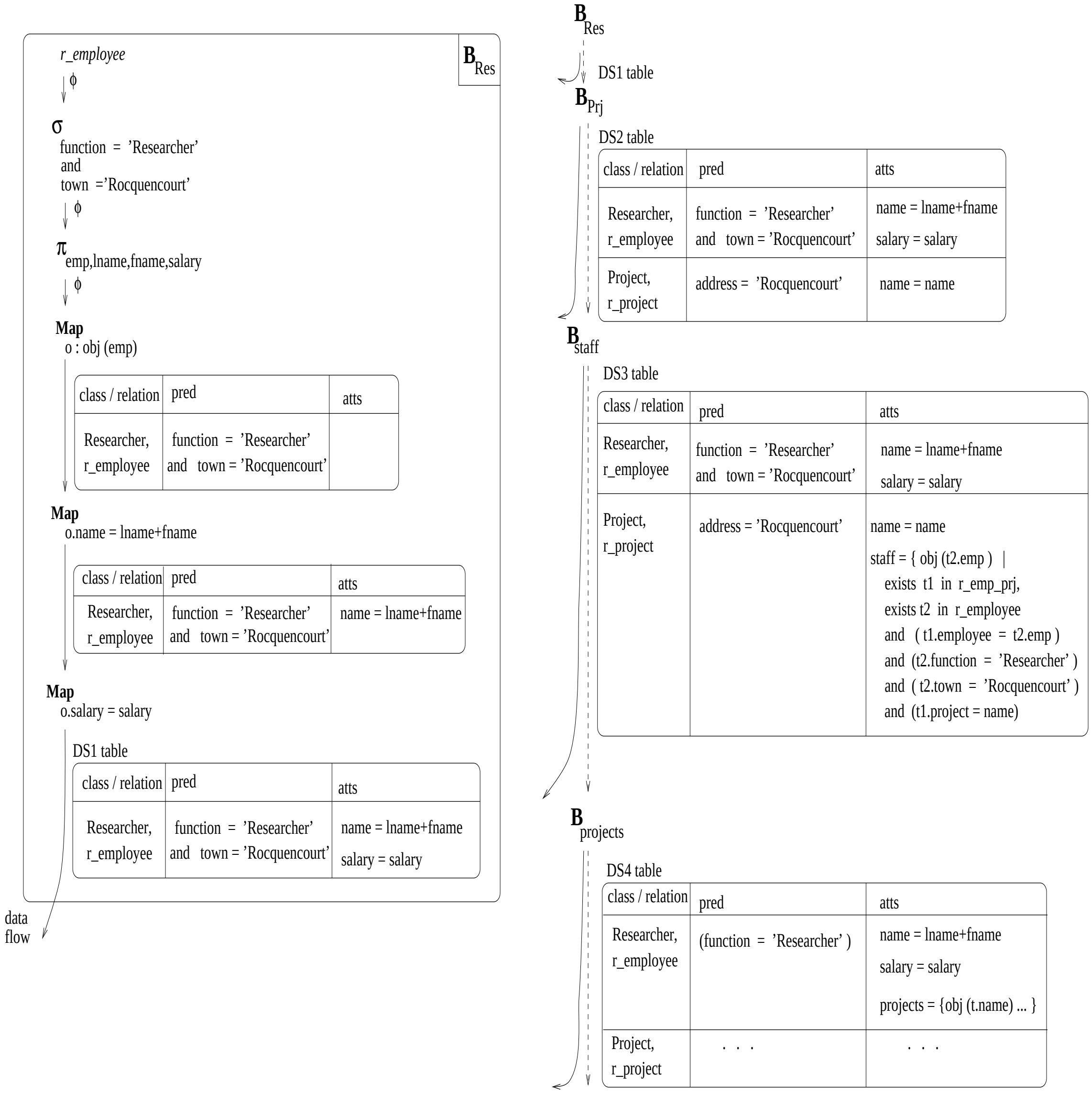

Figure 4: *Adding the DS Parameter*

DS allows us to define a whole new class of equivalences, called DS-equivalences ($\equiv_{DS}$), that rely primarily on side-effects. To illustrate this, consider removing the projection operation ($\pi_{emp,lname,fname,salary}$) from block $\mathbf{B}_{\text{Res}}$. This would result in a different algebraic data flow. However, this would not modify the object database state and should thus be considered as a correct rewriting. This is what DS-equivalences are for. We will see more involved examples of DS-equivalences including rewriting examples that will spread across several blocks in Section 4.4. For the moment, just remark that they are less constraining than usual algebraic ones (denoted $\equiv$). Given two arbitrary algebraic expressions e_1 and e_2, we have $e_1 \equiv e_2 \Rightarrow e_1 \equiv_{DS} e_2$ but not the opposite. Some examples of DS-equivalences can be found in [5].

4.3 Parameters of the Rewriting Process

Although the ultimate goal of the rewriting phase is optimization, some constraints that influence the communication or processing time have to be taken into account. These constraints are related to load balancing (e.g., the relational site should not be overloaded), and physical data organization as specified by the user. We discuss here the parameters that will guide the rewriting phase, i.e., the constraints and the resources we want to optimize. We also study the default representation in view of these parameters. Next, we consider some rewriting examples offering alternative tradeoffs between constraints and optimization. Section 5 shows one possible implementation of an optimizer relying on these parameters.

Where should we process? The relational/object migration process we consider, consists of three phases. (i) A query is evaluated by the relational system and its result is sent to the migration component, (ii) eventually some non database processing follows there, (iii) finally transactions are launched on the object system to create the database. This is illustrated on Figure 5. Note that an alternative solution would put the migration component at the relational site. From an optimization point of view, the only difference would be in communication costs.

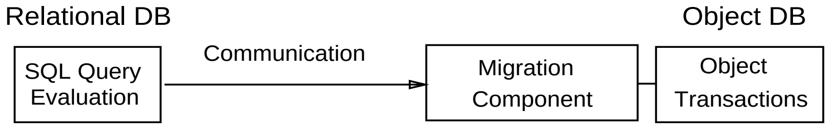

Figure 5: *The Processing Three Phases*

Given this architecture, we can find different ways to decompose a migration process. For instance, a possible and rather natural decomposition of blocks $\mathbf{B}_{\text{staff}}$ and $\mathbf{B}_{\text{projects}}$ would be as follows: *Map* operations are computed by the object transactions, *Group* operations are performed locally (in the migration component) and all other operations are performed at the relational site. Alternatively, we could perform all operations but the *Map*(s) in the migration component or migrate all relations in the object system and perform all processing there. There are of course many other alternatives. The choice of a solution depends on various factors: the availability of the two database systems, the cost of each operation in the different systems, the ability of the migration component to store and manage large amount of data, the cost and reliability of communication.

The decision to favour processing on one of the components rather than another has direct consequences on the rewriting process and important influences on the global migration cost. Indeed, if we decide to push as many operations as possible to the relational site, the rewriting should separate non-relational (e.g., grouping containing sets) and relational operations. Alternatively, performing all processing out of the relational system may increase the global migration cost since we will not be able to use relational optimization techniques, such as existing indexes. However, this strategy may sometimes be interesting since it minimizes the load of both database systems and shortens communications.

When should we process? Relational and object systems may be available at some times and unavailable at others. Thus, it is important to be able to slice the process into pieces that are run whenever and for as long as appropriate. To achieve that, the generated transaction programs are parameterized according to (i) time or (ii) size (number of created objects). The user can then launch a program by giving a fixed time or size. There are mainly two ways to decompose a program.

- Either, load all relations in the migration component, perform all processing there and then allow the user (or a program) to launch a sequence of object transactions. As was explained earlier, this strategy does not benefit from relational optimization technique and may imply an important cost overhead.

- Or, generate queries that load only the relevant part of data for a given transaction (i.e. decompose a large query into smaller ones). The advantage of this solution is that, as opposed to the previous one, it is flexible in terms of load balancing. Processing can be performed at any site(s). Of course, it also has a cost that must be taken into account by the optimization process: e.g., n small joins may be less efficient than a large one.

Physical database organization. As was explained in Section 3, we want to be able to specify ordering and clustering criteria in order to store objects in a manner adequate to further processing. Let us consider the following cluster statement:

`class` Project `cluster on` staff(Researcher)

In order to implement this cluster, the object system should at least be aware of the number of researchers in a project before creating it so as to be able to reserve the appropriate space. Even better, having all information about projects and related researchers at the same time would allow to store objects through one sequential access to the corresponding database file.

Now, let us consider the default representation. It creates researchers, then projects, before materializing the `staff/projects` relationship. Obviously, the system is in no position to guarantee the appropriate physical property without re-organizing its storage. Thus, the expression should be rewritten so as to consist of a merge of blocks $\mathbf{B}_{\mathrm{Prj}}$ and $\mathbf{B}_{\mathrm{staff}}$ first, then of block $\mathbf{B}_{\mathrm{Res}}$, or, even better in terms of object secondary access, of a merge of all three blocks $\mathbf{B}_{\mathrm{Res}}$, $\mathbf{B}_{\mathrm{Prj}}$ and $\mathbf{B}_{\mathrm{staff}}$ into one (see Figure 6).

As another example, the following statement specifies that projects should be split into two parts in order to allow different strategies for network or non-network related projects:

`class` Project `if` topic='Network' `cluster on` staff

Finally, specifying an order as in:

`class` Project `order by` name

implies the addition of a *Sort* operation in block $\mathbf{B}_{\mathrm{Prj}}$.

We will see later on how merging, splitting and ordering can be achieved (Figures 6, 7 and 8). For the present, let us just comment on the influence of these operations on the global migration cost and, accordingly, on the rewriting process. Consider once again the possibility of merging all three blocks $\mathbf{B}_{\mathrm{Res}}$, $\mathbf{B}_{\mathrm{Prj}}$ and $\mathbf{B}_{\mathrm{staff}}$ and thus creating projects, researchers and the `staff/projects` relationship at the same time. Again, this can be done in the migration component, independently from the database systems, or we can rely on the relational system to perform a join between the three relations and provide all relevant information (i.e., all `Project` and `Researcher` attributes). Note that by requiring the relational system to compute this large join, we may reduce the processing cost (less iterations on source relations, potentially faster join processing). However, note also that this will probably (i) increase the global communication cost (the large join can be much bigger than the sum of two selections plus a small join on two key attributes), (ii) the cost of the grouping operation (that will have to be performed on a larger set), (iii) the cost of local buffering, as well as (iv) the cost of a transaction decomposition.

Optimization. So far, we have been mainly concerned by migration constraints and have studied their influence on the migration process. Let us now consider optimization techniques.

The migration process involves (i) communications, (ii) local buffering, and (iii) relational, local and object processing, all of which should be optimized.

The easiest and most efficient way to reduce communication time is to perform all selections and projections at the relational site and only these operations (no joins). Then, the result is loaded in the migration component that may perform all further algebraic processing or may rely on the object system to do it. In terms of rewriting, this has few implications: the only requirement is to keep selections and projections in the upper part of the program trees (i.e. not mix these operations with the others). However, this may affect the processing time considerably (since we will not be able to rely on existing relational indexes to perform joins) and requires potentially large local buffering. The search strategy and cost model should take this into account. For instance, according to application requirements, communication costs can have a more or less important weight in the cost model.

The migration component may have less disk space than the database systems. Thus, local buffering should be taken into account by the rewriting process, either as a weighted component of the cost model (as is the case for the RelOO prototype) or as part of the search strategy. The migration component is in charge of storing the object/relational correspondence tables as well as all temporary results. If we want to maintain the consistency between an updatable relational database and its object counterpart, all correspondence tables must be maintained. However, if this is not the case, there exist ways to eliminate some of these tables. An example will illustrate this later on (see Figure 6). Another way to reduce local buffering is to avoid local processing altogether, or at least minimize it. As we will see, this is what we did in the RelOO prototype where the migration component performs only grouping operations on "reasonable" amount of data.

Some operations of the migration programs are purely object (database update operations). Others can be performed by either the object system or the migration component (grouping involving sets). Finally, some can be performed by any of the three migration actors (e.g., selections). We briefly explain the main optimization techniques applying to these four different kinds of operations.

- The only way one can optimize object operations during the rewriting phase is by taking object input/output into account. As we will see, some algebraic expressions involve random accesses to the object store (see the default representation) while others allow a sequential one (see Figure 6). By considering this in the cost model, we will be able to favour one or the other.

- Access to correspondence tables can and should be minimized. This can be done by merging blocks together as will be illustrated in the sequel.

- If no clustering is required, grouping operations can be avoided altogether and replaced by iterative invocations of set insertions at the object site. This, of course, may be more costly in terms of input/output operations. But, as we will see, we may sometimes rely on a correct ordering of the objects on disk and in the correspondence tables to minimize this cost (see Figure 8).

- The algebra we propose supports rewriting rules as found in [38, 11] to which rules relying on DS equivalences have been added. By using the first ones, we may rewrite all algebraic operations in the usual way and also use semantic information to reduce processing cost. For instance, if we know that all "Rocquencourt" researchers belong to a "Rocquencourt" project, we can remove the join with the *r_project* relation from both blocks $\mathbf{B}_{\mathrm{staff}}$ and $\mathbf{B}_{\mathrm{projects}}$ of the default representation. In terms of optimization, DS equivalences mainly allow reordering operations, adding some that may be beneficial to the migration process (e.g., sort before grouping, project before loading), and merging blocks in order to reduce the number of times one has to iterate over a relation as well as random access to objects.

This will be illustrated in the sequel. Another rewriting technique consists in materializing subexpressions common to two distinct blocks. This rewriting decreases communication and relational processing times but increases local buffering. Also, if the target systems support multi-threading or parallelism, DS equivalences can be used to highlight interesting parallelization strategies by organizing the algebraic blocks in an appropriate manner.

To Summarize... Many factors must be taken into account by the rewriting process, some of which are contradictory. A migration process may be inefficient in terms of global cost (i.e., sum of all processing costs) and still be considered highly interesting because it minimizes communication cost, or relational/object processing, or local storage maintenance, or is easily decomposable, or favours interesting clusters, etc.

For instance, let us consider again the default algebraic expression. Its main advantages are that it reduces communication cost and facilitates transaction decomposition since it cuts the process into small blocks. Furthermore, by memorizing the result of join operations locally (factorization), we can further reduce communication as well as processing time. Nevertheless, (i) processing time is bad because of the many accesses to the relational/object tables and (ii) it cannot support the cluster specifications given above.

Our rewriting technique considers all possibilities. This will now be illustrated through three examples. We will see in Section 5 an implementation of the technique based on simple heuristics and cost model.

To conclude, note that as mentioned in the beginning of this section, although the rewriting techniques are presented here for relational to object migration, the ideas go beyond that scope. In particular, one would find similar parameters to guide the optimization when migrating ASCII files to relational or object systems. This would probably imply that no operation be computed at the source site since file systems do not perform database operations.

4.4 Three Rewriting Examples

A detailed presentation of the rewriting rules that we use would be tedious. Instead, we illustrate them by three examples of rewriting on the default migration representation. In these examples, we also introduce splitting, clustering and ordering related to user-defined physical properties.

Merging All Four Blocks. The first rewriting merges all fours blocks into one and creates all objects and their relationships. It is illustrated in Figure 6. Some facts are noteworthy. (i) The join operations have been factorized and transformed into outer-joins. Outer-joins guarantee that we do not lose researchers or projects that do not appear in the relationship but must nonetheless be created. If we know that all projects and researchers are related, the outer-joins can be replaced by joins (semantic equivalence). (ii) We have removed the *Group* operation related to the `projects` attribute and rely on set insertion (+ = assignment) and the `(Researcher,r_employee)` correspondence table (accessed by $obj(emp)$) to update it. Researchers are created and updated in an embedded *Map* operation. (iii) We have grouped all *Map* operations into one. (iv) We have introduced a *Sort* operation on the projects name.

Let us now study block $\mathbf{B}_{\text{all}}$ using the migration parameters introduced above. This expression offers five positive characteristics. (1) If we are not interested in maintaining correspondences between relational and object entities (e.g., we are migrating archives), we may remove the correspondence table related to `Project` objects. However, the `(Researcher,r_employee)` table has to be maintained. Indeed, a researcher may belong to several projects and therefore may be

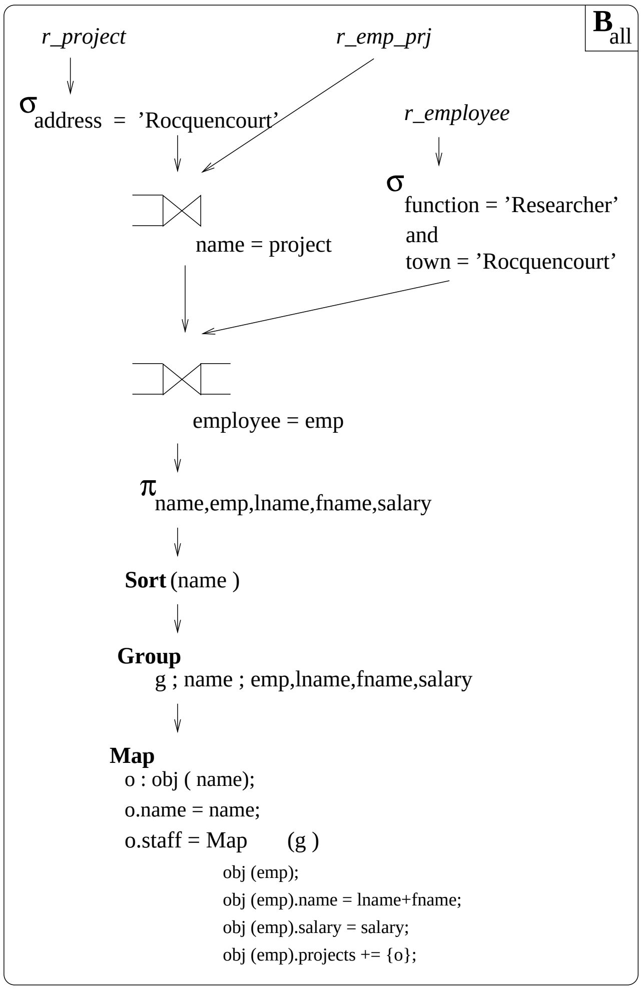

Figure 6: *Merging All Four Blocks*

accessed several times by $obj(emp)$. (2) We have reduced the number of joins and iterations to be performed on source relations. (3) We make no assumption on the process localization: e.g., we can compute the joins at the relational site, locally in the migration component or in the object system. (4) Assuming an incremental loading inside the object database, we maintain the referential integrity of the `staff/projects` relationship: i.e., at all times, a project referencing a researcher is referenced by the researcher. (5) The expression allows a better support of the following user specification:

`class` Project `order by` name `cluster on` staff(Researcher)

Now, let us consider the negative aspects of this rewriting. (1) The *Group* operation is performed on a very large set, implying large local storage and expensive computation. (2) If we perform the joins at the relational site, the global communication cost is much higher than that of the default representation. Furthermore, (3) cutting the loading from relational to object site (transaction decomposition) will be difficult and potentially expensive.

Splitting and Merging. The second rewriting is illustrated on Figure 7. Block $\mathbf{B}_{\mathrm{Res}}$ has been merged with the subpart of blocks $\mathbf{B}_{\mathrm{Prj}}$ and $\mathbf{B}_{\mathrm{staff}}$ related to "Network" projects. Thus, at the end of block $\mathbf{B}_{\mathrm{P/R/staff}}$, all researchers have been created, as well as all "Network" projects along with their `staff` attribute. The remaining part of the $\mathbf{B}_{\text{Prj}}$ and $\mathbf{B}_{\text{staff}}$ blocks follow, the expression ends with an unmodified block $\mathbf{B}_{\text{projects}}$.

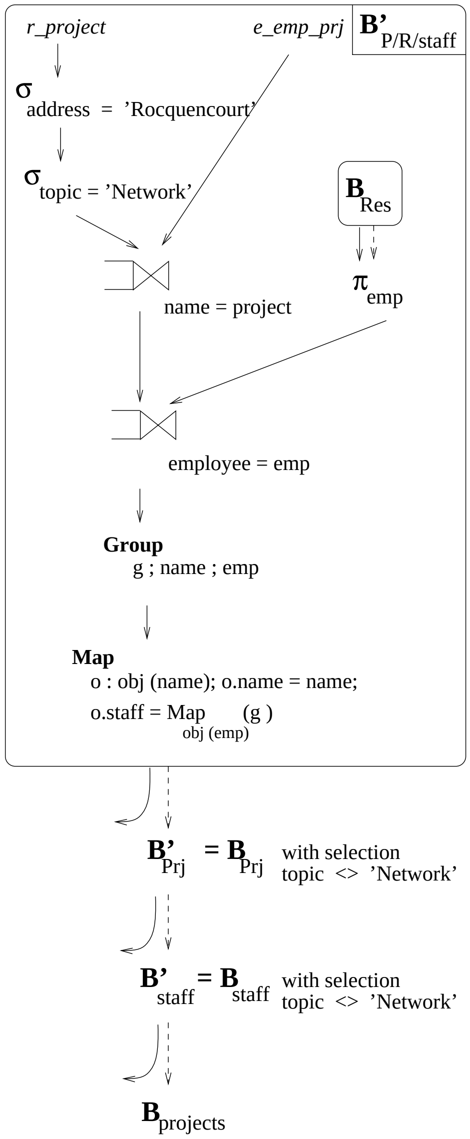

Figure 7: *Splitting According to Some User Specification*

The expression represents one possible way to take the following user cluster statement into account:

`class` Project `if` topic='Network' `cluster on` staff

Note that we can rewrite the three last blocks in any possible way and still keep that property. On the plus side, block $\mathbf{B}_{\text{P/R/staff}}$ features a smaller *Group* operation than the previous expression (only keys are grouped) and potentially smaller communications. Furthermore, there is no need to maintain a table of correspondence for "Network" related `Project` objects. On the negative side, it forces to perform a join outside of the relational system ($\sqsupset\!\bowtie_{employee=emp}$). Because of this, it also involves a larger buffering at the object site.

Playing With Group Operations. Our last example (see Figure 8) shows, among other interesting features, how one can push a *Group* operation to the relational site by adding an

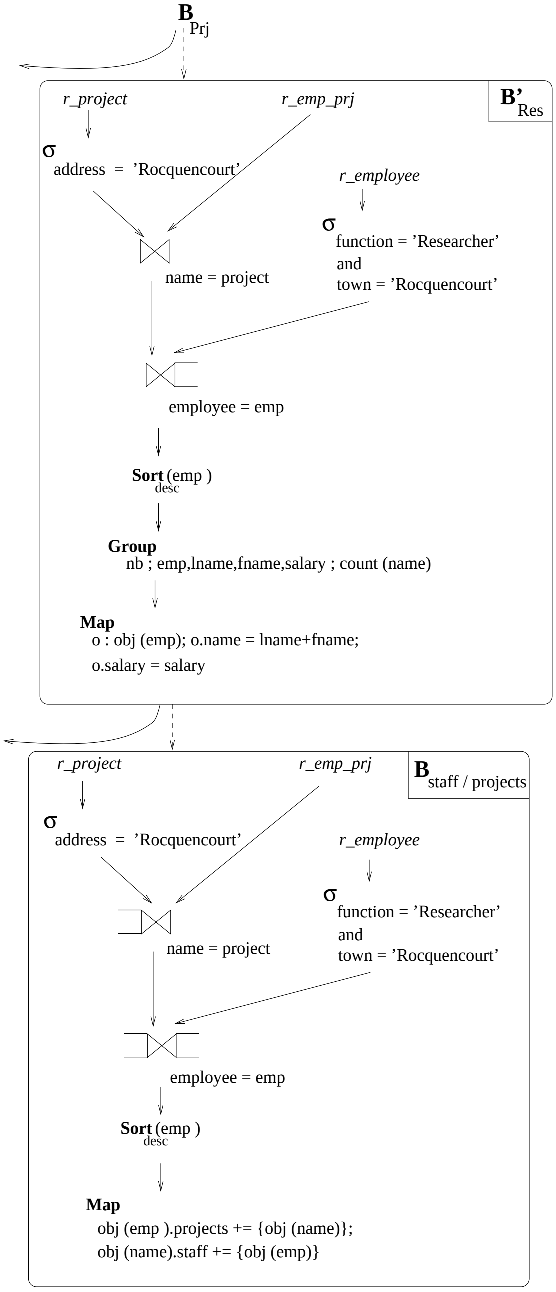

Figure 8: *Introducing Relational Grouping*

aggregate function. The expression consists of three blocks: the unmodified $\mathbf{B}_{\mathrm{Prj}}$ that takes care of `Project` objects, $\mathbf{B'}_{\mathrm{Res}}$ that creates researchers and $\mathbf{B}_{\mathrm{staff/projects}}$ that materializes the relationship between researchers and projects.

In block $\mathbf{B'}_{\mathrm{Res}}$, a *Sort* operation is added to order the researchers according to their key number. This *Sort* should also simplify the *Group* operation that follows and associates to each researcher the number of projects it belongs to. Finally, researchers are created through a *Map* operation. Although we do not have all information concerning the projects to which a researcher belongs, we know the size of the corresponding `projects` attribute. Thus, assuming the following cluster statement, we are in a position to organize the object storage in an appropriate manner:

`class` Researcher `order by desc` emp `cluster on` projects

As was stated above, the $\mathbf{B}_{\text{staff/projects}}$ materializes the relationship between researchers and projects. For this, it relies on the correspondence tables related to projects and researchers and on $+=$ assignments. A *Sort* has been added on relation *r_employee* and its effect will be explained later on.

Let us now see the positive and negative aspects of this expression. As was stated above, the expression allows (i) to support a cluster statement and (ii) push the *Group* operation on the relational site (since an aggregate function *count* is applied). Assuming that the relational system is more efficient than the migration component or the object system, this latter characteristic is interesting. If we want to avoid putting extra load to the relational system, this is still interesting since the grouping can be performed more efficiently (no need to create sets). Furthermore, using some rewriting, we could then buffer the result of the join and sort operations in the migration component and avoid evaluating them twice.

Unfortunately, this rewriting entails a more important processing at the object site and the need to support and access many times two correspondence tables (block $\mathbf{B}_{\text{staff/projects}}$). However, note that researchers (which should constitute the larger table) have been sorted on their key at creation time and before we materialize the relationship. Thus, we can rely on a sequential access to the table and to the object store (if it reflects the order of creation) to avoid unnecessary input/output operations.

5 The RelOO Migration Tool

The ideal migration tool should (i) work for any relational or object system and (ii) take full advantage of the rewriting possibilities we introduced in the previous section.

The first criterion is somehow hard to reach in conjunction with the second one. This is mainly due to the relative youth of the object systems as compared to their relational counterparts. Whereas a protocol such as ODBC [24] makes it possible to communicate in an efficient and homogeneous fashion with most relational systems, there is yet no object equivalent. The ODMG [10] standard OQL query language does not support update primitives and the various ODMG programming interfaces lack the declarativity necessary to abstract oneself from the target system. Although some object systems provide an ODBC interface, they cannot be viewed as a reasonable solution for generating object databases.

The second criterion requires a sophisticated architecture with a parameterized search strategy and cost model. The search strategy should make use of the various parameters introduced in Section 4 and, for instance, discard some sequence of equivalences or favour some other, given the user specifications and the target systems possibilities (e.g., what operations they can process). The cost model should mix communication, relational and object processing costs with appropriate application-specific weight factors. Not surprisingly, this kind of architecture resembles that of a mediator (e.g., [29, 36, 2]). In this paper, we present a reasonable solution relying on a fixed search strategy and a parameterized cost model. It extends a previous prototype that followed user requirements but had little optimization capacities [3]. We briefly present the optimizer and code generator of the RelOO system that is currently under development in the VERSO group at INRIA. RelOO targets are any ODBC compliant systems on the relational side and the O_2 system on the object side.

5.1 The RelOO Optimizer

The RelOO optimizer is being developed in Java[4]. Its inputs are (i) the default algebraic representation given in Section 4, (ii) the user constraints on physical organization and (iii) the cost model. It works in two phases:

1. According to the user constraints, sorting and selection operations are added. For instance, the ordering instruction given in Section 3 on class `Project` adds a *Sort* operation and the clustering instruction related to "Network" projects adds two selections ($\sigma_{topic='Network'}$ and $\sigma_{topic\neq'Network'}$, see Figure 7). Then we apply a first class of DS-equivalences whose goal is to merge and order blocks according to the user-given cluster specifications. This class contains equivalences that reorder or merge blocks, push *Map* operations and combine *Map* and *Group* operations.

   This first phase returns a new algebraic representation where all objects that should be clustered together are created, as much as possible, through a single and appropriate *Map* operation. Blocks and operations in this representation are partially ordered (i.e., some blocks precede others according to the order on clusters and some operations inside a block cannot be moved).

2. The second phase applies all equivalences on the result of the first, respecting the partial ordering and relying on the cost model to choose the best solution. The equivalences are standard algebraic equivalences plus DS-equivalences: that of the first class and some more allowing the removal of unions (this has not been illustrated in the paper), the introduction of a projection or an appropriate sort (if it does not invalidate a previous one) or the removal of *Group* operations. Since blocks are never un-merged (i.e., merge equivalences are applied in only one direction) and the partial ordering is not broken, the resulting expression implements the user-given physical constraints (if it is coherent).

The cost model we implemented relies on simple statistics and considers (i) the result size of the "relational" expression (for communication cost), (ii) the size and number of the relational attributes involved in sequences of join operations (to ease transaction decomposition), (iii) the size of the relational/object correspondence tables, (iv) the cost of accessing the object store and the correspondence tables (sequential versus random accesses), (v) the size required by local buffering and the computation cost of the various operations according to their execution context. For instance, by giving an infinite cost to a relational operation (i.e., *Selection*, *Projection*, *Join*, *Sort* and *Group* with aggregates on source relations) following a non-relational one, we guarantee that all relational processing is performed at the relational site. This is motivated by the fact that relational systems should be more efficient to process queries on source relations since they usually maintain indexes and can eventually rely on integrity constraints to optimize the evaluation. In the same manner, since *Map* operations can be performed only on the object site, we minimize object processing by giving an infinite cost to a *Group* following a *Map*. In this way, *Group* operations can be performed by the migration component. Finally, we minimize local buffering by assigning an infinite cost to a group operation following another one. This cost model guarantees that each block consists of (i) a sequence of relational operations, (ii) possibly one *Group* operation that will be evaluated locally in the migration component and (iii) some *Map* operations that will be evaluated on the object site.

[4]Note that the time devoted to the optimization process is of little consequence since it is performed only once and can be seen as part of the specification process that may take a long time to be achieved.

5.2 The RelOO Code Generator

Once the rewriting process is over, the RelOO system constructs one transaction program per block, each program being parameterized by either time or size. We briefly describe here how the time criteria is taken into account during the code generation. The size criterion is a sub-case of the time criterion.

We do not explain the translation process from algebra to relational, local or object operations. Although rather intricate, this part is done using standard techniques. As explained above, the optimization generates blocks with three distinct parts. The upper part is relational and is translated into an SQL query. Then comes a *Group* operation, if any, which is translated into a local Java program. Finally, the lower part consists of *Map* operations that are translated into appropriate O_2 code. The generated programs are then given to the RelOO migration control environment which controls their ordered execution and maintains the necessary correspondence tables. It also provides access functions to these tables that may be invoked by O_2 transactions.

Time-parameterized programs cut the object transactions according to a given duration. Since we want (i) to avoid multiple evaluations of the same query and (ii) to minimize local storage whenever possible (i.e., when no materialization is needed for factorization purposes), this kind of program requires that we construct a "reasonable" SQL query. By reasonable, we mean that it should return a number of tuples that corresponds to the time allocated to the object processing. For this, we work in two stages. First, relying on an average time of object creation and the number of creations in a block, we compute the expected size of the SQL query. Then, given the size of the source relation, its minimal and maximal key values and assuming uniform distribution of key values, we infer an SQL query. This is obviously not a perfect solution and further work should be done to improve this (e.g., using histograms as found in [31]). The query is then evaluated. Processing follows in the migration component. Then a transaction is launched on the object database. The transaction program loops until it has no more data to process or the elapsed time is greater than the input time. Then, it commits and returns the number of tuples that were processed. The remaining tuples are stored locally in order to be taken into account by the next program invocation.

6 Conclusion

We presented a framework allowing flexible and efficient migration of legacy relational data into object databases. The framework consists of (i) a declarative language to express database transformations and physical data organization and (ii) an algebra-based optimization technique that takes into account user-given constraints.

The problem of the automatic generation of an object schema given a relational one has been largely addressed. Our work is complementary. We believe that the framework we provide can support, in an efficient manner, the various solutions that have been proposed.

We have already developed a tool [3] that takes into account user-given constraints and are currently implementing a complete optimization module and a migration control environment in Java on top of the O_2 database system. The plan is to validate the tool on real applications while working on the following topics:

- Parallelization of blocks: what are the conflicts on the relational and object sides (simultaneous accesses to correspondence tables and object store)?
- The support of relational updates: how can we use a relational log to propagate updates

from the relational database to the object one?

- Physical optimization: could we unburden the object system by creating objects directly in the object store?

Acknowledgments:

We are deeply grateful to Serge Abiteboul from the Verso Group at INRIA, to Dave Maier from the Oregon Graduate Institute, to Guy Ferran and Didier Tallot from O_2Technology and to Tova Milo from Tel Aviv University for many fruitful discussions.

References

[1] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 238–247, 1991.

[2] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[3] S. Amer-Yahia. From Relations to Objects: The RelOO prototype. In *ICDE conference – Industrial session*, Birmingham, England, April 1997.

[4] S. Amer-Yahia. *The complete RelOO syntax.* France, 1998. http://www-rocq.inria.fr/verso/Sihem.Amer-Yahia.

[5] S. Amer-Yahia, S. Cluet, and C. Delobel. Rewriting with side-effects. Technical report, INRIA – Verso Group, France, 1997. http://www-rocq.inria.fr/verso/Sihem.Amer-Yahia.

[6] T. Atwood, D. Barry, J. Duhl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann, San Francisco, 1994. http://www.odmg.org.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System — The Story of O_2.* Morgan Kaufmann, San Mateo, California, 1992.

[8] A. Behm, A. Geppert, and K.R. Dittrich. On the Migration of Relational Schemas and Data to Object-Oriented Database Systems. In *Proc. of Intl. Conf. on Re-Technologies in Information Systems*, Klagenfurt, Austria, December 1997.

[9] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. of Intl. Conf. on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.

[10] R.G.G. Cattel, D. Barry, D. Bartels, M.Berler, J. Eastman, S. Gammerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, 1997. http://www.odmg.org.

[11] S. Cluet and G. Moerkotte. Classification and Optimization of Nested Queries in Object Bases. In *Bases de Données Avancées*, pages 331–349, 1994.

[12] S. Davidson and A.S.Kosky. WOL: A Language for Database Transformations and Constraints. In *Proc. of Intl. Conf. on Data Engineering*, pages 55–65, UK, April 1997.

[13] C. Delobel, C. Souza dos Santos, and D. Tallot. Object Views of Relations. In *Proceedings of the 2nd International Conference on Applications of Databases - ADB '95*, San José, California, December 1995.

[14] F. Exertier. ROBIN: Generating Object-Oriented and WEB Interfaces for Relational Databases. Technical report, Bull, January 1997. White paper.

[15] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas acoording to ODMG-93. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Singapore, December 1995.

[16] J. Fong. Converting Relational to Object-Oriented Databases. *SIGMOD Record 1997*, 26(1), March 1997.

[17] Gemstone. *http://www.gemstone.com*.

[18] Jean-Luc Hainaut. Transformation-based Database Engineering. In *Proc. of Intl. Conf. on Very Large Data Bases*, Zurich, Switzerland, 1995. Tutorial.

[19] Tadashi Iijima, Shirou Udoguchi, and Motomichi Toyama. Applying Database Publishing Technology to the Migration from a Relational Database to an Object-Oriented Database. Technical report, Information and Computer Science - Keio University, 1998.

[20] Persistence Software INC. Bridging the Gap between Relational Data and Object Oriented Development. Technical report, Persistence Software INC., September 1993.

[21] J. Jahnke, W. Schäfer, and A. Züendorf. A Design Environment for the Migration of Relational to Object-Oriented Database Systems. In *ICSM*. IEEE Computer Society, 1996.

[22] I. Kwan, J. Fong, and Q. Li. A Methodology for Reenginnering Relational to Object Oriented Database Systems via an Extended E-R Model. In *Proc. of the Intl. Conf. on Data and Expert Systems Applications (DEXA)*, pages 152–161, UK, September 1995.

[23] Franck Lebastard. *DRIVER: Une couche Objet Virtuelle Persistante pour le Raisonnement sur les Bases de Données Relationnelles.* PhD thesis, Institut National des Sciences Appliquées, Lyon, France, March 1993.

[24] Microsoft. *Open DataBase Connectivity software development kit manual – version 2.0*, 1994. http://www.ddodbc.com.

[25] R. Missaoui, J.M. Gagnon, and R. Godin. Mapping an Extended Entity-Relationship Schema into a Schema of Complex Objects. In *Object Oriented and Entity Relationship*, Australia, December 1995.

[26] O2Technology. *O2DBaccess User Manual Release 4.6*, September 1996.

[27] Objectstore. *http://www.odi.com*.

[28] Oracle. *http://www.oracle.com*.

[29] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[30] Hewlett Packard. OpenODB: Facilitating Change. Technical report, Hewlett Packard, September 1991.

[31] V. Poosala, Y. Ionnidis, P. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 294–305, Montreal, Canada, June 1996.

[32] S. Ramanathan and J. Hodges. Reverse Engineering Relational Schemas to Object-Oriented Schemas. Technical Report MSU-960701, Mississipi State University, Mississipi, 1996.

[33] S. Ramanathan and J. Hodges. Extraction of Object-Oriented Structures from Existing Relational Databases. *SIGMOD Record 1997*, 26(1), March 1997.

[34] E. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, Vancouver, 1992.

[35] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1991.

[36] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Proc. of Intl. Conf. on Distributed Computing Systems*, volume 26(1), March 1996.

[37] M. Tsangaris and J. Naughton. On the Performance of Object Clustering Techniques. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 144–153, USA, June 1992. ACM.

[38] J. D. Ullman. *Database and Knowledge base systems.* Computer Science Press, 1988.

[39] VMark. *http://www.o2tech.fr.*

[40] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 30–41, Zurich, Switzerland, 1995.

[41] J. L. Wiener and J. F. Naughton. Incremental Loading of Object Databases. Technical report, University of Wisconsin, 1996.

[42] J.L. Wiener. *Algorithms for Loading Object Databases.* PhD thesis, University of Wisconsin - Madison, 1995.