On the Optimization of Queries Containing Regular Path Expressions

Andreas Henrich[1] and Stefan Jamin[2]

[1] Otto-Friedrich-Universität Bamberg, Fakultät Sozial- und Wirtschaftswissenschaften, Praktische Informatik, D-96045 Bamberg, Germany,
Andreas.Henrich@sowi.uni-bamberg.de
[2] CENIT AG Systemhaus, Schulze-Delitzsch-Str. 50, D-70565 Stuttgart, Germany,
S.Jamin@cenit.de

Abstract. One of the main characteristics of object-oriented database management systems is the explicit representation of relationships between objects. A simple example for a query addressing these relationships arises, if we assume the object types *Company*, and *Division* with the relationship *has_division* from *Company* to *Division*. In this case a query might ask for the companies which have a division called *"strategy"*. The query might start with the companies and navigate to the divisions which can be reached via the *has_division* relationship. Finally the query has to check if the *name* attribute of the *Division* object is *"strategy"*. Since there is no direct condition for the companies in the query, this query execution will be costly. If we assume that there is a reverse relationship *division_of* from *Division* to *Company*, an alternative execution plan might start with the *"strategy"* divisions and follow this reverse relationship. In this case an index structure for the *name* attribute of the *Division* objects can be exploited to speed up query processing.
In the present paper we describe a query optimizer which exploits this potential invertibility of navigational operations in queries. Our approach is based on, but not limited to the context of the ISO and ECMA standard PCTE and P-OQL.

1 Introduction

In contrast to relational database management systems object-oriented database management systems (ooDBMS) allow for the explicit representation of the relationships between the maintained objects. The various ooDBMS differ mostly in the expressive power of their modeling facilities for these relationships. Some systems allow only special attributes which consist of a set of objects. Other systems provide the means of a link to represent a relationship between objects. Furthermore some systems provide different link categories to represent different types of relationships, or they allow for the application of key- and/or non-key-attributes to the relationships.

Such relationships are often addressed in queries to the database. Assume for example an object base with the object types *Student*, *Course* and *Lecturer*

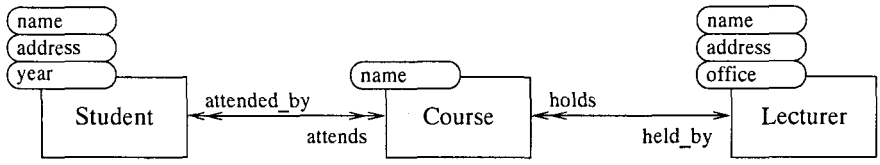

Fig. 1. Simple example schema with relationships

and the relationships *attends* from *Student* to *Course* and *held_by* from *Course* to *Lecturer*. This schema is illustrated in figure 1.

On this object base we could e.g. ask for the names of all students which attend a course held by lecturer *"Smith"* together with the corresponding course names. The straight forward way to evaluate this query would be to address all *Student* objects stored in the database and to check for each student if there is a path consisting of an *attends* and a *held_by* link referencing a lecturer named *"Smith"*. Under the natural assumption that there are far less lectures than courses, and far less courses than students, this is obviously an extremely costly operation. If there are, on the other hand, reverse relationships for the relationships *attends* and *held_by* — as in our example schema — another query processing plan might start with the lecturers. Only for those lecturers with the name *"Smith"* the students attending a course held by the lecturer would have to be addressed in this case. Especially in the case where an index for the *name* attribute of the lecturers exists this query processing plan would be much more efficient than the first one.

To exploit this potential invertibility of navigational operations in queries the query optimizer should consider query execution plans based on such inverted navigational operations in addition to the conventional query optimization techniques. To this end, the query optimizer has to determine invertible path expressions and to select the most beneficial inversion. Thereby the query optimizer should consider the available index structures and select the structure (or structures) which should be applied.

We have developed such a query optimizer for the P-OQL query language. P-OQL [8, 9] is an OQL-oriented query language for the object management system of PCTE [19], which in turn is the ISO and ECMA standard for an open repository [17, 18]. The environment consisting of PCTE and P-OQL is extremely challenging for the sketched optimizer facility, because the data model of PCTE contains extremely powerful facilities for the representation of relationships. The relationships can have key and non-key attributes, and there are different categories of relationships. The query language P-OQL allows to define navigational operations by regular path expressions addressing the attributes and the categories of the relationships and providing various iteration facilities.

In the following we first describe the essential concepts of PCTE and P-OQL (c.f. section 2). Thereafter we give various examples for the inversion of regular path expressions in P-OQL, which might allow for a more efficient query pro-

cessing. In section 4 we present the architecture of our query optimizer. Section 5 deals with the cost estimation techniques employed to choose from the different possible query formulations and in section 6 we present some experimental results. Finally section 7 gives a short discussion of related approaches and section 8 concludes the paper.

2 Example Environment

2.1 PCTE

As mentioned PCTE (*Portable Common Tool Environment*) is the ISO and ECMA standard for a public tool interface (PTI) for an open repository [17–19]. As one of its major components PCTE contains a structurally object-oriented object management system (OMS) designed to meet the special requirements of software engineering environments.

The data model of PCTE can be seen as an extension of the binary Entity-Relationship Model. The object base contains objects and relationships. Relationships are normally bi-directional. Each relationship is realized by a pair of directed links, which are reverse links of each other. The type of an object is given by its name, a set of applied attribute types, and a set of allowed outgoing link types. New object types are defined by inheritance.

A link type is given by a name, an ordered set of attribute types called key attributes, a set of (non-key) attribute types, a set of allowed destination object types, and a category. PCTE offers five link categories: *composition* (defining the destination object as a component of the origin object), *existence* (keeping the destination object in existence), *reference* (assuring referential integrity and representing a property of the origin object), *implicit* (assuring referential integrity) and *designation* (without referential integrity).

Throughout this paper we will use the schema given in figure 2 as the basis for our examples. It consists of the object types *Student*, *Course*, *Employee*, *Thesis* and *Project*. The attribute types applied to each object type are given in the ovals at the upper left corner of the rectangle representing the object type.

The link types are indicated by arrows. A double arrowhead at the end of a link indicates that the link has cardinality *many*. Links with cardinality many must have a key attribute. In the example the numeric attribute *no* and the string attribute *problem* are used for this purpose. For example the link type *attends* from *Student* to *Course* has such a key attribute and is hence described as *"no.attends"*. Therefore an instance of this link type can be addressed by its *link name* which consists of the concrete value for the key attribute and the type name separated by a dot – e.g. *"3.attends"*. Link types with cardinality *one* do not need a key attribute and are given in the schema by a dot followed by the link type name. An *'E'* or *'R'* in the triangles at the center of the line representing a pair of links, indicates that the link has category *existence* or *reference*.

Finally the schema contains the link type *has_advisor* as an example for a link type with a non-key attribute. In our case this is the attribute *meeting*.

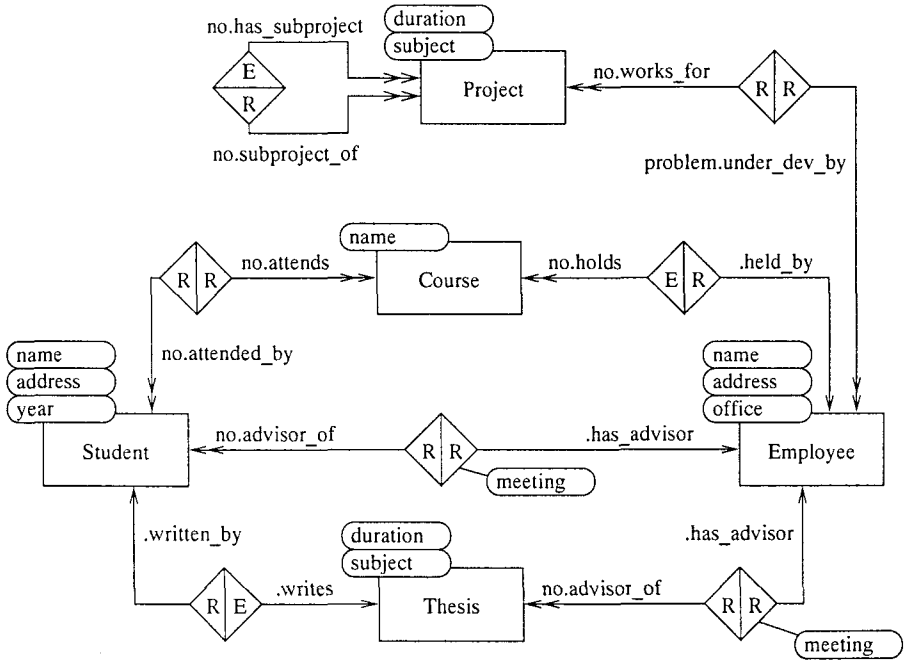

Fig. 2. Example schema

With respect to the query optimizer presented in this paper some further characteristics of the PCTE data model have to be stressed: (1) The reverse link type of a link type is non-ambiguous. I.e. each link type has exactly one reverse link type and a link type can be the reverse link type for only one link type — except for *designation* links. (2) A link type may have multiple destination object types which may or may not be leaves of the inheritance hierarchy. (3) Since links with category *designation* are an exception of the rule that each link has a reverse link a navigation traversing a *designation* link cannot be inverted.

2.2 P-OQL

P-OQL [8] is an OQL-oriented query language for PCTE, and OQL (*Object Query Language*) [2] is the ODMG proposal for a query language for object-oriented database management systems. The main differences between P-OQL and standard OQL are due to the adaptation to the data model of PCTE. Hence, especially the treatment of links is specific to P-OQL.

A query in P-OQL is either a select-statement, or the application of an operator (like *sum*).

Assume that we search for pairs with the name of a student and the name of an employee, where the student attends a course held by the employee and the student is in the fourth year. The following query in P-OQL yields these pairs:

```
select   A:name, B:name
  from   A in Student, B in (A:_.attends/.held_by/->.)
  where  A:year = 4
```

In the **from-clause** of this query two base sets are defined: Base set A addressing all students in the object base and base set B addressing all objects which can be reached from the actual object of base set A via a path matching the regular path expression "A:_.attends/.held_by/->.". In this path expression the prefix "A:" means that the actual element of base set A is used as the starting point of the definition. "_.attends/.held_by" means that exactly one link of type *attends* and one link of type *held_by* must be traversed. The underscore "_" is used as a wildcard for numerical key attributes denoting that arbitrary key values are allowed. In addition intervals can be specified for numerical key attributes and regular expressions can be used for string key attributes. Since there is no key attribute defined for the link type *held_by*, the notation ".held_by" is used to specify that a *held_by* link has to be traversed. The notation "/->" is used in the regular path expression "A:_.attends/.held_by/->." to address the destination object of the path. In addition "->" can be used to address the last link of the path. The dot "." at the end of the regular path expression means that the object under concern is addressed. It is also possible, to address an attribute or a tuple of values (see [8] for more details).

Due to the definition of an *independent* base set (base set A) and a *dependent* base set (base set B), each student is combined with each employee, which can be reached from the object representing the student via a path matching the regular path expression.

Altogether, a base set can be defined in P-OQL in five different ways: (1) giving an object type name or an object type name suffixed by a "^" meaning that objects of all subtypes are addressed as well; (2) using a link type name to address all links of a given type; (3) defining a set of objects or links using a regular path expression, as with base set B in our example; (4) defining a set of objects or links via a sub-select; or (5) passing a set of objects or links via the API (*application programming interface*) when submitting a query.

In the **where-clause** of our example query the considered combinations of students and employees are restricted to those for which the *year* attribute of the student has the value *"4"*. Besides such simple conditions P-OQL for example allows the use of quantifiers and subqueries in the where-clause.

The **select-clause** of the example query states that a multiset of pairs is requested. Each pair consists of the name of the student and the name of an employee who holds the course the student attends.

An additional facility of P-OQL — which is relevant to our optimization problem — allows for the iteration over one or more links of the same type. Assume for example that we are interested in the subprojects of a project with subject *"Workflow"*. These subprojects can be determined in P-OQL as follows:

```
select   A:subject, A:[_.has_subproject]+/->subject
  from   A in Project
  where  A:subject = "Workflow"
```

Here the regular path expression "A:[_.has_subproject]+/->subject" is used to create a (multi-)set with the subjects of all subprojects of the project addressed in base set A. The meaning of "[_.has_subproject]+" is that one or more links matching the link definition "_.has_subproject" have to be traversed. Alternatively P-OQL knows the iteration facilities "[*path_definition*]*" to indicate that zero or more paths matching *path_definition* have to be traversed and "[*path_definition*]" to indicate that a path matching *path_definition* is optional. After traversing an arbitrary number of *has_subproject* links, the *subject* attribute of the destination object is addressed using "/->subject".

In addition to the link definitions used in the above example, which have been based on a given link type name and a definition of the allowed values for the key attributes, P-OQL allows the specification of a set of link categories instead, meaning that all links having one of the given categories fulfill this link definition. E.g. the expression "[{c,e}]+/->." addresses all objects which can be reached via a path consisting only of links with category *composition* or *existence*. Furthermore, not only the category of the link itself, but also the category of its reverse link can be specified using *"@"* as a prefix for the category.

Finally it has to be mentioned that the ODMG standard OQL also permits the definition of reverse links and the use of path expressions for the definition of base sets. However, since the ODMG data model does not include attributes for links or link categories, the path expressions in OQL can be seen as a special case of the regular path expressions in P-OQL.

3 Inverting Regular Path Expressions

3.1 Simple Cases

First let us consider a slightly extended version of the example query given in section 2.2. In addition we require that the address of the employee is "Cologne":

select A:name, B:name
from A *in* Student, B *in* (A:_.attends/.held_by/->.)
where A:year = 4 *and* B:address = "Cologne"

In the first base set this query addresses all students and in the second base set the corresponding employees are addressed starting from the student actually under concern via the regular path expression "A:_.attends/.held_by/->.". In this way each student is combined with each employee which can be reached via a path matching the path expression. Furthermore, due to the semantics of P-OQL, this means that when a student attends two courses held by the same employee, this student/employee pair is considered twice. However, the same result can be achieved as well when the query addresses all employees and inverts the regular path expression "A:_.attends/.held_by/->." in order to address the corresponding students in a second base set. In this way we yield the equivalent P-OQL query:

select A:name, B:name
from B *in* Employee, A *in* (B:_.holds/_.attended_by/->.)
where A:year = 4 *and* B:address = "Cologne"

To see that both queries are equivalent, first we have to note that the data model of PCTE assures that for each path matching the regular path expression "_.attends/.held_by/->." there is a reverse path matching the regular path expression "_.holds/_.attended_by/->.". Hence, if an employee e is reached from a student s via a path matching "_.attends/.held_by/->.", there is always a reverse path from e to s matching "_.holds/_.attended_by/->.". If there are multiple paths from a student s to an employee e, we have exactly the same number of reverse paths. As a consequence, we have the same pairs in the base sets of both queries and each pair has exactly the same number of occurrences.

If there is an index for the attribute *address* of the object type *Employee* which is more selective than the index for the attribute *year* of the object type *Student*, the second query will lead to a more efficient query execution plan.

The above example might suggest that an inversion of a regular path expression used to define a base set can always be performed easily without any effects on the select-clause and the where-clause of the query. Unfortunately this is not true. The situation becomes much more complicated whenever one of the following cases occurs: (1) The values of the key attributes are restricted in the regular path expression. (2) One of the iteration facilities of P-OQL is used. (3) The destination object type of a link is ambiguous. (4) A link definition in the regular path expression is stated by the allowed link categories. In the following sections we describe how these more complicated cases can be handled.

3.2 Conditions for Link Key Attributes

As mentioned in section 2.2 there are different facilities to restrict the allowed key attribute values for traversed links. In the following query we use two of these facilities in order to navigate from the student addressed in base set A to some specific projects. Furthermore we require that the student should be in the fourth year and that the duration of the project should be at least 36 month:

```
select   A:name, B:subject
  from   A in Student, B in (A:[2..4].attends/.held_by/1.works_for/->.)
  where  A:year = 4 and B:duration >= 36
```

Due to the conditions for the link key attributes which are integrated in the regular path expression we cannot invert the path expression "A:[2..4].attends/.held_by/1.works_for/->." in one step. Rather we have to split up the expression when inverting it, in order to address the key attributes of the reverse links, for which the conditions have to be enforced. In consequence, this means that the conditions which are integrated in the original regular path expression are moved into the where-clause of the inverted query:

```
select   A:name, B:subject
  from   B in Project, H1 in (B:*.under_dev_by->.),
         H2 in (H1:/.holds/_.attended_by->.), A in (H2:/.)
  where  A:year = 4 and B:duration >= 36
         and H1:@no = 1 and H2:@no >= 2 and H2:@no <= 4
```

In the inverted query we first navigate from the project under consideration to all outgoing *under_dev_by* links [1]. These links are addressed in the auxiliary base set H1. In the where-clause we have to assure that the key attribute value of the reverse link of the link addressed in base set H1 is *"1"*. In P-OQL the *"@"*-sign can be used to switch from a link to its reverse link. Hence, the condition "H1:@no = 1" assures that the value of the key attribute of the reverse link is *"1"*. Starting with base set H1 we then navigate to the *attended_by* links which are addressed in base set H2. The first "/" in the regular path expression "H1:/.holds/_.attended_by->." used for this purpose specifies that we navigate from the current link in base set H1 to its destination object. Starting at this destination object we then traverse a *holds* links and reach the *attended_by* link. For the reverse links of these *attended_by* links the condition "H2:@no >= 2 *and* H2:@no <= 4" assures that the key attribute value is in the interval [2..4]. Finally the base set definition "A *in* (H2:/.)" addresses the destination objects of these links in base set A.

3.3 Iteration Facilities

For each employee living in Cologne the following example query calculates a set with the subjects of the projects with a duration of at least 36 month he works for. To this end the query iteratively follows the *subproject_of* links:

select A:name, B:subject
from A *in* Employee, B *in* (A:_.works_for/[_.subproject_of]*/->.)
where A:address = "Cologne" *and* B:duration >= 36

The inversion of this regular path expression does not cause major problems, since we can use analog iteration facilities in the inverted version:

select A:subject, B:name
from B *in* Project, A *in* (B:[_.has_subproject]*/*.under_dev_by/->.)
where A:address = "Cologne" *and* B:duration >= 36

This alternative query formulation exploits that for each path p_1 matching the expression "_.works_for/[_.subproject_of]*/->." there is a reverse path p_2 matching the expression "[_.has_subproject]*/*.under_dev_by/->.". Unfortunately the situation becomes much more complicated as soon as the iteration facilities are combined with conditions for the key attribute values.

3.4 Iterations Facilities and Conditions on Link Keys

The following query directly corresponds to the query considered in section 3.3 except that we traverse only the first link of type *subproject_of* for each project — that means we traverse only links with the key attribute value *"1"*:

[1] Recall that the notation "->." at the end of a regular path expression addresses the last link of the path, whereas the notation "/->." addresses the destination object. Further note that the "*" is used in "B:*.under_dev_by->." to allow arbitrary string key attribute values.

```
select   A:name, B:subject
  from   A in Employee, B in (A:_.works_for/[1.subproject_of]*/->.)
  where  A:address = "Cologne" and B:duration >= 36
```

If we combine the solutions presented in the previous sections for conditions on link keys and for iteration facilities in a straight forward manner to invert the regular path expression "A:_.works_for/[1.subproject_of]*/->.", we yield:

```
select   A:name, B:subject
  from   B in Project, H1 in (B:[_.has_subproject]*->.),
         A in (H1:/*.under_dev_by/->.)
  where  A:address = "Cologne" and B:duration >= 36 and H1:@no = 1
```

Unfortunately this query is not equivalent to the original query, because the condition "H1:@no = 1" is checked only for the last link of each path matching the regular path expression "B:[_.has_subproject]*->.".

Therefore, we have to apply a different approach in situations where conditions for the key attribute values occur inside an iteration. To this end, we recall the basic aim of the query inversion. This aim is to apply an index structure for the destination object type of the original path expression. Let us assume for example that there is no index for the *address* attribute of the employees. Then in the above example query the base set definition "A *in* Employee" means that we have to scan all employees. On the other hand, there might be an index structure for the *duration* attribute of the projects. If we assume that the condition "B:duration >= 36" is rather restrictive, there will be only few employees working in such projects. Therefore it might be useful to consider not all employees, but only the employees working in such projects. The objects representing these employees can be determined by the following select statement:

```
select distinct H2:.
  from   H1 in Project, H2 in (H1:[_.has_subproject]*/*.under_dev_by/->.)
  where  H1:duration >= 36
```

Two aspects have to mentioned with respect to this query: (1) We use a *select distinct* here to avoid duplicates in the result. (2) The query will in general return more employees than actually needed, because the conditions for the link key attributes given in the original query are not reflected here.

Now we simply use this query instead of the object type *Employee* to define base set A in the original query:

```
select   A:name, B:subject
  from   A in (select distinct H2:.
                 from   H1 in Project,
                        H2 in (H1:[_.has_subproject]*/*.under_dev_by/->.)
                 where  H1:duration >= 36 ),
         B in (A:_.works_for/[1.subproject_of]*/->.)
  where  A:address = "Cologne" and B:duration >= 36
```

Using this query an index structure for the *duration* attribute of the projects can be employed to speed up query processing.

3.5 Ambiguous Destination Object Types

Another problem caused by the inversion of regular path expressions, is the existence of ambiguous destination object types for some link types. In the schema given in figure 2 the link type *advisor_of* is such a link type with two destination object types (*Student* and *Thesis*). So by inverting a regular path expression it may be necessary to address objects by using such a link type, but not all destination object types should be considered. The following example uses the link type *has_advisor* to define the dependent base set B:

select A:name, B:name
from A *in* Student, B *in* (A:.has_advisor/–>.)
where B:address = "Cologne"

If we invert the regular path expression "A:.has_advisor/–>.", we have to use the link type *advisor_of*, but only destination objects of type *Student* should be addressed. To this end, we can use a type test predicate of P-OQL:

select A:name, B:name
from B *in* Employee, A *in* (B:_.advisor_of/–>.)
where B:address = "Cologne" *and* A:. *is of type* Student

Similar situations can arise due to inheritance. To illustrate such a situation, we extend our university schema given in figure 2 by two subtypes for the object type *Employee* as shown in figure 3. As usual, in PCTE a subtype t_1 of an object type t_0 inherits the applied attribute types and the allowed outgoing link types. Furthermore, if t_0 is defined as the destination object type of a link type, links of this type can as well point to objects of type t_1, because an object of type t_1 can be used wherever an object of type t_0 is required.

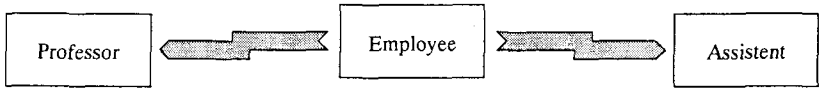

Fig. 3. Two subtypes of the object type *Employee*

To see the effects of inheritance for the inversion of regular path expressions, we reconsider the example query given in section 3.1 combined with the inheritance situation given in figure 3:

select A:name, B:name
from A *in* Student, B *in* (A:_.attends/.held_by/–>.)
where A:year = 4 *and* B:address = "Cologne"

Now the inverted query has to address the object type *Employee* and its subtypes in the first base set. To this end, we can use the notation *ObjectType*^ which addresses an object type and his descendant object types:

```
select   A:name, B:name
  from   B in Employee^ , A in (B:.holds/_.attended_by/->.)
  where  A:year = 4 and B:address = "Cologne"
```

3.6 Link Categories Used in Regular Path Expressions

The last feature of P-OQL we want to consider here, is the definition of the allowed link categories in a regular path expression. This feature is extremely useful when combined with the document retrieval facilities included in P-OQL [9]. Unfortunately we cannot present these facilities here due to the space limitations. As a consequence, the following example may seem relatively artificial.

Assume the following query starting with projects and following *reference* links, which in this case lead to employees:

```
select   A:subject, B:name
  from   A in Project, B in (A:{r}/->.)
  where  B:address != "Cologne"
```

Since there is only one link type with category *reference* originating from the object type *Project* we can invert this query as follows:

```
select   A:subject, B:name
  from   B in Employee, A in (B:{@r}/->.)
  where  B:address != "Cologne" and A:. is of type Project
```

Here the regular path expression "B:{@r}/->." enforces, that the reverse link of the traversed link has the category *reference*. Since this condition is true for multiple outgoing link types of the object type *Employee*, we have to add the condition "A:. *is of type* Project" in the where-clause of the query.

In general the situation is a bit more complicated, because there will be multiple destination object types for links with the required category. In this case the query optimizer has to look for a unifying supertype or it can build the union of the objects of the various types which can be accessed via sub-selects.

4 The Optimizer Architecture

In section 3 we have given various examples for situations where a query can be inverted to use another object type as the starting point for query processing. In the present section we will describe the architecture of our query optimizer which tries to exploit such inverted query formulations to speed up query processing.

The query optimizer is implemented as a preprocessor. This preprocessor receives a query in P-OQL syntax and it returns a query in P-OQL syntax with some extensions specifying the index structures to be used. The different steps performed by our optimizer are shown in figure 4. The solid lines with the numbers (1) to (7) represent the main control flow whereas the dashed lines identified by the letters (a) to (e) represent the information transfer between those components of the optimizer which provide additional information for the optimization process. The main components can be described as follows:

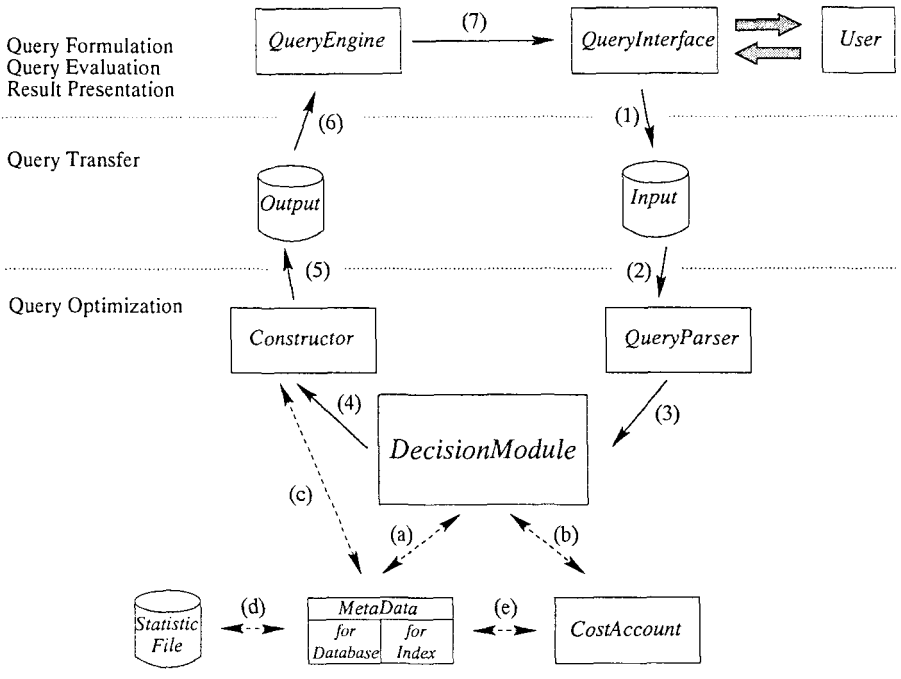

Fig. 4. The optimizer architecture

QueryInterface: The *QueryInterface* stores the P-OQL query in the *Input*-file (1) and expects the result for presentation to the user (7).

QueryParser: This component parses the original query from the *Input* (2) and creates an internal representation which is passed to the *DecisionModule* (3). Thereby the *QueryParser* checks whether there are base set definitions using regular path expressions. In case, the corresponding base sets are marked.

DecisionModule: The *DecisionModule* analyses the base sets of the actual query to detect interdependent base sets. For each group of interdependent base sets the definitions of the dependent base sets are scanned in order to evaluate whether they can be inverted. When the definition of a dependent base set e.g. includes a complex sub-select or a navigation over a *designation* link, an inversion is regarded as impossible. On the other hand in the cases explained in section 3 an inversion is possible. For each invertible base set definition the destination object type is determined. These destination object types and the object type addressed in the independent base set of the original query formulation built a set with object types which can be used as starting points of the query formulation. For these alternatives the query execution costs are estimated based on the information provided by the *CostAccount* module (b). For this cost estimation information about the cardinality of the base sets, information about the available index structures,

and information about the conditions in the where-clause of the query is considered. The decision for the alternative with the lowest expected costs is thereafter passed to the *Constructor* module (4).

Constructor: This module actually performs the inversion of the query. To this end, the *Constructor* depends on information from the *MetaData* module (c), e.g. to determine the name of a reverse link type.

MetaData: This module provides an interface to retrieve the relevant meta data for the database and the index structures.

StatisticFile: To assure a minimum of information even if there exists no index structure for an object type, this file contains e.g. an estimation of the number of instances of each object type.

CostAccount: The *CostAccount* module computes the relative costs of the original query plan and the potential alternative query plans by using the available information from the MetaData module (e). The details of this module are discussed in section 5.

QueryEngine: This module evaluates the final query by means of a nested-loop approach. For the base sets for which an index structure has been selected the index structure is employed. The result of the evaluation is sent to the QueryInterface (7), which presents it to the user.

5 Cost Estimation and Index Selection

For the discussion of the applied cost estimation technique it is important to mention that our implementation uses a multi-dimensional index structure. This index structure is presented in detail in [11, 10]. For the present paper it suffices to know that this index structure allows to address multiple attributes in one access structure [2]. A corresponding index definition might e.g. look as follows:

Index for object type Student: *1.* name, *2.* year, *3.* address

Since the index structure is symmetric, the same selectivity is provided for all supported dimensions. Furthermore the index structure is well suited for situations where multiple attributes are addressed in the where-clause of a query, because in these cases the selectivity for all supported attributes is combined.

In this context some administrative information is maintained for each index structure: (1) an *object counter* containing the number of instances of the indexed object type, (2) the *number of dimensions* of the index structure, (3) the indexed attributes, and (4) an *object type hierarchy* specification, defining whether the index is created only for objects of the defined object type itself, or for objects of the defined type and all descendent types.

Based on this information for each object type j which could be used as the starting point of the query formulation the existing index structures are

[2] It should be obvious that the use of this multi-dimensional index structure is by no means a prerequisite for the application of our query optimization approach. If the approach is used with one-dimensional index structures — like the B-tree [5] — only the formulas presented in this section have to be slightly adapted.

determined. Now two cases have to be distinguished: (1) For none of the object types an index exists: In this case we use the information from the *StatisticFile* to select the object type with the fewest instances as the starting point of the query formulation. (2) Otherwise we always choose to apply an index structure. To select this index structure we proceed in two steps: In the first step we try to determine the *"best"* index structure for each object type j, for which index structures exist. In the second step we select the object type which should be used as the starting point of the query formulation.

1. step: This step is performed for each object type which could be the starting point of the query and for which at least one index structure is defined with an *object type hierarchy* specification corresponding to the query.
For the index structures of object type j we proceed as follows:

1. We collect the conditions from the where-clause which refer to object type j. We denote the number of these conditions by q_j.
2. For each index i we count those conditions, which can be supported by the index. We denote this number by $h_{j,i}$.
3. For each index i of object type j we calculate $r_{j,i} = \frac{h_{j,i}}{q_j}$ (the share of the conditions for object type j which are supported by index i).
4. Now we assume that the index structure with the highest value for $r_{j,i}$ is the *"best"* index structure for object type j. If there are multiple index structures with this highest value, we choose the index which has the fewest dimensions. If this is not unique as well, we choose an arbitrary index out of these index structures. Let i_j^* denote the chosen index structure for object type j.

2. step: Now we can choose the object type j which should be used as the starting point of the query formulation: For each object type j we calculate the value $C_j = \textit{object counter} \times \max(0.01, 1 - r_{j,i_j^*})$, where *object counter* is taken from the administrative information of the index structure i_j^*. Since small values for *object counter* and high values for r_{j,i_j^*}, which always is a value out of $[0,1]$, are desirable, we use $1 - r_{j,i_j^*}$ in the formula. We use $\max(0.01, 1 - r_{j,i_j^*})$ because otherwise this factor would be zero for $r_{j,i_j^*} = 1$, i.e. when all conditions are supported by the index structure. So to assure that the *object counter* can always influence the formula we use $\max(0.01, 1 - r_{j,i_j^*})$. Finally the *DecisionModule* chooses the object type j with the lowest value for C_j as the starting point of the query formulation.

6 Experimental Results

In this section we present some experimental results achieved with our optimizer. The tests were performed on a Sun Sparc20 with two processors and 96 MB of main memory. The database for the tests was filled with synthetic data created for the university schema given in figure 2 and extended in figure 3. Table 1 summarizes the existing index structures for the object types *Student* and *Employee* which occur in our example queries.

First we examined the example query explained in section 3.1:

Table 1. Index structures defined for the tests

object type	object counter	index number	number of dimensions	attributes	object type hierarchy
Student	6040	1	2	name, term	this type only
Student	6040	2	3	name, address, term	this type only
Student	6040	3	2	name, address	this type only
Employee^	1995	1	2	name, address	with subtypes
Employee^	1995	2	3	name, address, office	with subtypes
Employee^	1995	3	1	office	with subtypes

select A:name, B:name
from A *in* Thesis, B *in* (A:_.attends/.held_by/->.)
where A:year = 4 *and* B:address = "Cologne"

For this query the optimizer returned the following query, where the base set definition "B *in* #1# Employee^ " specifies that index 1 has to be used to speed up the loop over the employees:

select A:name, B:name
from B *in* #1# Employee^ , A *in* (B:.holds/_.attended_by/->.)
where A:year = 4 *and* B:address = "Cologne"

Table 2. Performance results for the first test query

using the index structures for the object type Student		
index number	number of read index pages	relative execution time
1	225	72 %
2	192	66 %
3	188	66 %
using the index structures for the object type Employee^		
index number	number of read index pages	relative execution time
1	47	23 %
2	51	26 %
3	115	100 %

The performance results presented in table 2 show that the optimizer in fact selected the best possible query plan for our example query. Without the inversion of the query the execution time would have been nearly three times higher. So the optimizer selected the correct object type to start with and it selected the best possible index structure for this object type.

In the second test query we are searching for the subjects of all theses advised by an employee named *"Brubeck"*:

```
select   A:subject
  from   A in Thesis, B in (A:.has_advisor/->.)
  where  B:name = "Brubeck"
```

In this case the optimizer returned the following query:

```
select   A:subject
  from   B in #1# Employee^ , A in (A:_.advisor_of/->.)
  where  B:name = "Brubeck" and A:. is of type Thesis
```

Here the condition "A:. *is of type* Thesis" is inserted into the where-clause because of the ambiguous destination object types of the link type *advisor_of*.

Table 3. Performance results for the second test query

without using index structures for the object type Thesis		
index number	number of read index pages	relative execution time
—	—	100 %
using the index structures for the object type Employee^		
index number	number of read index pages	relative execution time
1	5	6 %
2	6	8 %
3	115	75 %

Table 3 shows the performance results for the second test query. Since there is no index structure defined for the object type *Thesis* we have included the time for the query execution without an index structure in the table. Again the optimizer chose the best possible query plan for our example query.

It remains to mention that we have performed various other tests as well. The optimizer selected the best possible plan in over 80 % of the cases, and whenever the selected plan was not the best one, it was only slightly worse.

7 Related Work

Although there is a great number of approaches for query optimization problems in the field of ooDBMS, to our best knowledge this is the first approach which tries to invert complex regular path expressions in order to exploit index structures defined for the destination object types of the paths.

For a general overview on query optimization problems in ooDBMS we refer to [13]. In [15] Mitschang presents the basic concepts and implementation aspects

of query processing and query optimization. Furthermore multiple interesting articles dealing with various aspects of query optimization can be found in [7]. Finally some rule-based approaches to query optimization are presented in [6], [14] and [1]. However, all these papers do not address topics related to the potential inversion of regular path expressions.

On the other hand some work dealing with path expressions and related aspects has been published, even though it is not directly concerned with the inversion of the path expressions.

In [4] Christophides, Cluet and Moerkotte extend an object algebra with two new operators and present some interesting rewriting techniques for queries featuring generalized path expressions. However, their approach does not address the problem of determining applicable access structures or aspects of the inversion of regular path expressions.

Ozkan, Dogac, and Evrendilek present a heuristic to determine the optimum execution order for the joins needed to process a path expression [16]. In contrast to our approach they assume that the references to objects are maintained as implicit joins which are converted to explicit joins during the optimization phase. Since we assume that the navigation via a link in PCTE is an extremely efficient operation supported by the physical structures of the OMS, these considerations are not applicable in our context.

Other approaches deal with the exploitation of materialized views for query processing [3] or with a general architecture for a query optimizer for OQL [12].

Summarizing, all the aforementioned approaches do not address the aspect that regular path expressions can be inverted in order to exploit index structures defined for the destination object types of the paths.

8 Conclusion and Future Work

In this paper we have presented an approach which exploits the potential invertibility of regular path expressions in order to apply index structures defined for the destination object types of the original path expressions. The approach has been described for the environment of PCTE and P-OQL, but it can be easily adapted for ooDBMS with simpler link concepts than PCTE.

Our future work will be concerned with an extension of the presented approach to other facilities of the query language. One important point in this respect are sub-selects, which can often be eliminated or flattened and processed in the same way as regular path expressions. Another aspect are quantifiers. Especially existence quantifiers seem to be well suited for an integration in our approach. Also improved cost estimation techniques are an interesting research area, and finally a formal framework for our optimization approach is needed.

References

1. L. Becker and R.H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, Juni 1992.

2. R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, Cal., USA, 1993.
3. C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Advances in Database Technology - EDBT'94. 4th Int. Conf. on Extending Database Technology, Proceedings*, volume 779 of *LNiCS*, pages 323–336, Cambridge, UK, 1994.
4. V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 413–422, Montreal, Canada, 1996.
5. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
6. J.C. Freytag. A rule-based view of query optimization. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 173–180, San Francisco, Cal., USA, 1987.
7. J.C. Freytag, D. Maier, and G. Vossen, editors. *Query Processing for Advanced Database Systems.* Morgan Kaufmann, San Mateo, Cal., USA, 1994.
8. A. Henrich. P-OQL: an OQL-oriented query language for PCTE. In *Proc. 7th Conf. on Software Engineering Environments*, pages 48–60, Noordwijkerhout, Netherlands, 1995. IEEE Computer Society Press.
9. A. Henrich. Document retrieval facilities for repository-based system development environments. In *Proc. 19th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 101–109, Zürich, 1996.
10. A. Henrich. A homogeneous access structure for standard attributes and document representations in vector space. In *Proc. 3rd Int. Workshop on Next Generation Information Technologies and Systems*, pages 154–161, Jerusalem, Israel, 1997.
11. A. Henrich and J. Möller. Extending a spatial access structure to support additional standard attributes. In *Proc. 4th Int. Symp. on Advances in Spatial Databases*, volume 951 of *LNiCS*, pages 132–151, Portland, ME, USA, 1995.
12. A. Heuer and J. Kröger. Query optimization in the CROQUE project. In *Proc. 7th Int. Conf. on Database and Expert Systems Applications*, volume 1134 of *LNiCS*, pages 489–499, Zürich, 1996.
13. Y.E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121 – 123, März 1996.
14. M.K. Lee, J.Ch. Freytag, and G.M. Lohmann. Implementing an interpreter for functional rules in a query optimizer. In *Proc. 14th Int. Conf. on Very Large Data Bases*, pages 218 – 229, Los Altos, Cal., USA, 1988.
15. B. Mitschang. *Query Processing in Database Systems (Anfrageverarbeitung in Datenbanksystemen).* Vieweg Verlag, Braunschweig, Wiesbaden, 1995. in German.
16. C. Ozkan, A. Dogac, and C. Evrendilek. A heuristic approach for optimization of path expressions. In *Proc. 6th Int. Conf. on Database and Expert Systems Applications*, volume 978 of *LNiCS*, pages 522–534, London, UK, 1995.
17. Portable Common Tool Environment - Abstract Specification / C Bindings / Ada Bindings. Standards ECMA-149/-158/-165, 3rd edition, 1993.
18. Portable Common Tool Environment - Abstract Specification / C Bindings / Ada Bindings. ISO IS 13719-1/-2/-3, 1994.
19. L. Wakeman and J. Jowett. *PCTE - The standard for open repositories.* Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1993.