Proceedings of the 4rd GI Workshop on Foundations of Database Systems, ed. U. W. Lipeck, and R. Manthey, European Computer−Industry Research Center Technical Report ECRC−92−13, Barsinghausen, Germany, June 1992

Query Optimization in COCOON

Christian Rich, Marc H. Scholl
Computer Science Department, University of Ulm
e−mail: {rich, scholl}@informatik.uni−ulm.de

1. Introduction

It is clearly crucial for the success of OODBMSs to find efficient implementations that improve on the performance of relational systems, rather than being powerful in terms of modelling and features, but just too slow to be used.

In this talk, we sketch the mapping of COCOON to DASDBS, our prototype storage manager. DASDBS was chosen as the storage system, because of its support for complex storage structures. First of all, the support of nested relations allows for the storage of hierarchically clustered data. That is, we have hierarchical access structures with an arbitrary level of nesting, as well as the opportunity to define nested join indices. This can be very useful to store COCOON objects and the "relationships" between them. Second, the DASDBS interface offers powerful data retrieval and manipulation operations. The system has a set oriented, algebraic interface, with efficient operations on complex objects.

The overall architecture of the COCOON implementation on top of the DASDBS kernel system is shown in Figure 1.1. The two aspects of this architecture, structure mapping and operations mapping, are realized in the physical design tool and the query optimizer, respectively. Physical design uses the COCOON schema, statistics about cardinalities and distribution, and a description of a transaction load to propose a good internal storage structure expressed in terms of nested DASDBS relations. At transaction processing time, the optimizer has to translate COOL operations down to operations on these physical NF^2 structures. The execution plans generated consist of physical NF^2−algebra operators, some of which, such as joins, are implemented in the high−level query processor, others are DASDBS kernel calls. In the sequel, we elaborate on each of these two aspects separately.

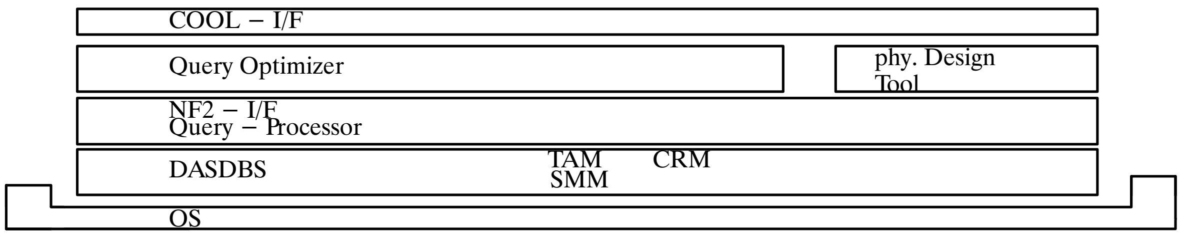

Figure 1.1: The COCOON − DASDBS Architecture

2. COCOON

Essentially, the COCOON model as described in [SS90] is an object−function−model. Its constituents are objects, functions, types and classes. COCOON offers a variety of structuring capabilities to model complex objects. Class and type hierarchies (i.e. classification and generalization), the abstraction concepts of aggregation and association, and derived methods offer a flexible modelling framework. The query language, COOL, offers object−preserving as well as object−generating generic query operators plus generic update operators. The key objective in the design of COOL was its set−oriented, descriptive characteristics, similar to a relational algebra. COCOON is a core object model, basically we have objects (concrete and abstract) and one type constructor, namely set. Other features can be added later due to the orthogonality of the language.

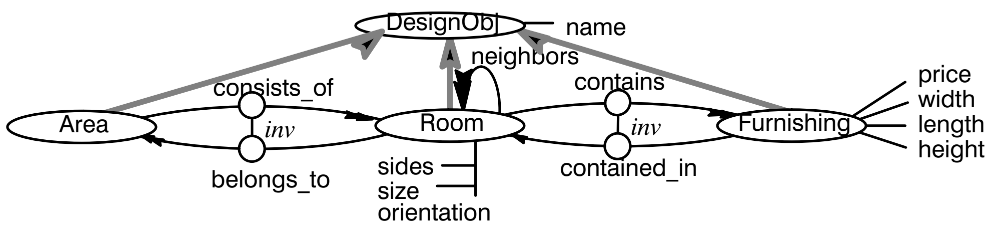

Figure 2.1: Example COCOON Schema

As an example, we will use an architectural design application. The COCOON graphical representation is given in Figure 2.1. In this application we are mainly dealing with rooms, furnishings and areas. All of them are design−

objects. Rooms are described by their name, orientation, position, size, etc. Between certain rooms there is a neighborhood−relationship. Each room contains a set of furnishings, which are described by their name, price, width, length, height, etc. Each room belongs to some areas, and each area consist of some rooms. That is, the room 'Office1' could be part of the area 'working−area', and may contain furnishings like 'Wooden Desk' and 'Computer'. A COOL query asking for offices, having no 'computer' is given below:

Q := select [name = "Office"
and select [name = "Computer"] (contains) = {}] (Room);

3. Physical Database Design

In this section we discuss the mapping of COCOON schemas to nested relations (DASDBS). That is, given a COCOON database schema, what are the alternatives for the internal DB layouts, and further, given a transaction load, which internal DB layout results in the least overall cost of transaction execution. This optimization task is performed by the physical design tool.

3.1 Alternatives for Physical DB Design

In order to explain the alternatives for mapping COCOON database schemas to nested relations at the physical level, we proceed by stepping through the basic concepts of the COCOON object model, and showing the implementation choices. Since the choices for each of the concepts combine orthogonally, this spans a large decision space that is investigated by the physical database design tool (next section).

Implementing Objects

According to the object−function paradigm of COCOON, an object itself is sufficiently implemented by a unique identifier (OID), that is generated by the system. All data related to an object in one way or the other will refer to this identifier (see below). We denote for each object type *A*, attributes of internal relations containing the OID of objects of type *A* by *AID*.

Implementing Functions

In COCOON, functions are the basic way of associating information (data values or other objects) to objects. In principle, we can think of each function being implemented as a binary relation, with one attribute for the argument OID and the other for the result value (data item or OID). In case of set−valued functions the second attribute will actually be a subrelation of unary subtuples, containing one result (OID or data value) each. So, in principle, each single−valued function as well as each multi−valued function would be implemented in a binary relation. For example, in our architectural design application, the functions *size* and *furnishings* of *Rooms* would result in two relations, *Room_size(RID, size)* and *Room_furnishings(RID, furnishings(FID))* respectively. Obviously, there are some choices. The decision space as far as function implementations are concerned includes the following alternatives in our current approach:

Bundled vs. Decoupled: Each function defined on a given domain object type might either be stored in a separate (binary) relation as shown above: the *decoupled* mode. Alternatively, we can *bundle* functions together with the relation implementing the type.
In the example above, the bundled implementation of *Room size* and *Room furnishings*, would yield the following type table : *Room(RID, size, furnishings(FID)).*

Logical vs. Physical Reference: A function returning a (set of) object(s), not (a) data value(s), can be implemented by storing just OIDs (*logical reference*) of result objects or by including a TID (*physical reference*) as well. Continuing on the above example (bundled), inclusion of physical references for the *neighbors* function, would result in: *Room(RID, size, furnishings(FID, @F)).* Notice the naming convention: (physical) reference attributes have @ as a prefix.

Oneway vs. Bothway References: A function can be implemented by a forward reference only (*oneway*), or it can be implemented with backpointers (*bothway*). For example, in case the *contained_in* function of *furnishings* would not be present in the conceptual schema, we could nevertheless decide to implement it, to have the *contains* function of *Rooms* supported with backpointers as well.

Reference vs. Materialized: Functions returning (possibly sets of) objects, not data values, can be implemented by the various forms of references discussed up to now. Alternatively, however, we can directly *materialize* the object−tuple(s) representing the result object(s) within the object−tuple representing the argument object. This is a way of achieving physical neighborhood (clustering). In our example, the decision to materialize the *furnishings* function of *Room* objects would generate a nested type table that contains the type table for *furnishings* as a subrelation: *Room(RID, size,... furnishings(FID,name, ...)).* Obviously, we need no backward references in this case. Furthermore, this alternative is free of redundancy only if the materialized function is *1:n*, that is, it's inverse is single−valued.

Computed vs. Materialized: Finally, an additional option is to materialize derived (computed) functions. Assuming that some function can be computed, we could nonetheless decide to internally materialize it, if retrieval dominates updates to the underlying base information significantly. The more retrieval dominates updates, and the more costly the computation is, the more likely is the case that materialization pays off. For example, the *size* function of *Rooms* is derived from the actual geometry of the Room. But computing the *size* incurs quite some effort and if object shapes rarely change, materializing the *size* function clearly is a good strategy (see also [KM90a,KM90b]).

Implementing Types, Classes, and Inheritance

The COCOON model separates between types and classes, this results in having two inheritance hierarchies: one between types (organizing structural, function inheritance), and one between classes (organizing set inclusion). Since classes are always bound to a particular (member−) type, physical design for types is the larger grain approach, whereas design for individual classes would be the finer grain approach (remember, there may be more than one class per type). Currently, we do the physical design on a type basis, that is, all objects of a given type are physically represented in the same way (even if they belong to several classes). Classes are implemented as views over their underlying type table. This results in the following choices w.r.t. types, classes, and inheritance:

Types: Each object type is mapped to a type table with at least one attribute, containing the OID. Additional attributes are present in case of any bundled functions and/or materializations of object functions. The type table *T* may itself be a subrelation of some other table, if type *T* was materialized w.r.t. a function returning *T*−objects.

Classes: Each class *C* is implemented as a view over its underlying type table. If the class is defined by a predicate ("all"−classes and views in COCOON), this predicate is used as the selection condition. If the class is defined to include manually added member objects ("some"−classes in COCOON), the underlying type table is extended by a Boolean attribute *C* that is set to true, if the object is a member of this class *C*.

Inheritance: Subtyping is implemented by having one type table per subtype. Three possibilities are considered:

- an object−tuple is included in each supertype's table. In case there are any bundled or materialized functions, these are not repeated in the subtypes' tables. In this case, object−tuples in subtype tables might optionally include physical references to supertype tuples.
- an object−tuple is included only in one type's table, that of the most specific subtype. In case of any bundled or materialized functions in supertypes, these are also included in the subtype's table.
- an object−tuple is included in each supertype's table. In case of any bundled or materialized functions in supertypes, these are also included in the subtype's table.

4. Query Optimization

In this section we discuss the transformation and optimization of queries that are given to the system in terms of the COCOON database schema. It is the task of the query optimizer to map these COOL queries down to the physical level by: (i) transforming them to the nested relational model and algebra as available at the DASDBS kernel interface, and (ii) select a good (if not the best) execution strategy.

Our proposal proceeds in two phases: the first one uses the information about the physical database schema for a transformation of the given COOL query into a nested relational algebra representation. The second phase following the transformation chooses the specific execution strategy, e.g. the ordering of operators, as well as the best implementation strategies. For example, whether a nested loop or a sort merge join is selected. This leads to the architecture shown in figure 4.1 of our query optimization process.

In the first step, the transformation task, we use a "Class Connection Graph". We will elaborate more on that in the following. The second step, that is, the optimization phase, we use a rule−based algebraic query optimizer, generated with the EXODUS Optimizer Generator [GD87].

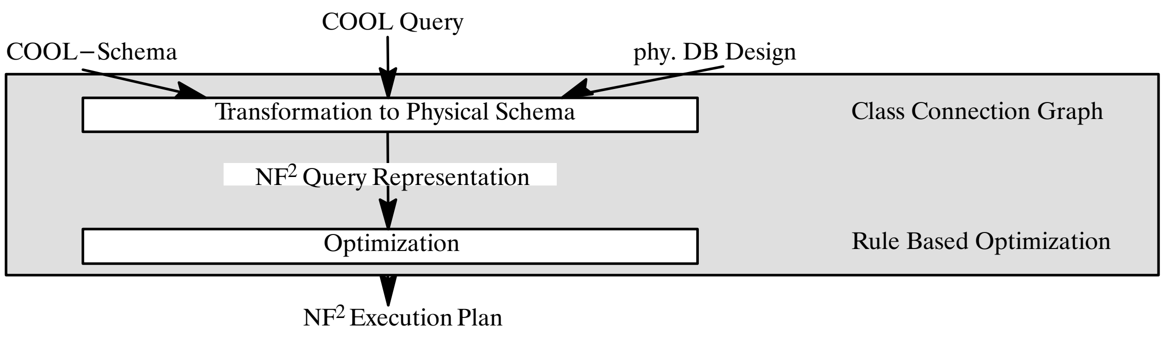

Figure 4.1: Optimizer Architecture

4.1 Transforming COOL−Queries onto the Physical Schema

The input to the optimizer is a COOL−query expressed on the conceptual schema, while the transformations apply to execution plans, i.e., on the physical schema. Therefore, we have to do a translation to an algebraic query representation on the physical schema first. To do this, we use a Class Connection Graph similar to the one proposed in [LV91].

After the input query is parsed, the optimizer scans at the same time the query graph (a graph representing the given query) and a physical schema graph. From these, the class connection graph is constructed. A possible physical schema graph is given in Figure 4.2. The nodes are files, which may implement

− a single class extension (e.g. A implements Areas and D implements DesignObjects)
− several class extensions (e.g. R implementing Room and Furnishings)
− a part of a class extension, when a class extension is vertically fragmented.

The arcs denote the kind of function implementation. Solid arcs denote materialized functions (e.g. contains). Dashed arcs are functions stored as references to subobjects inside the instance of the owner object.

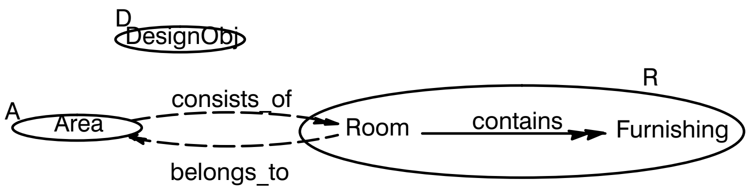

Figure 4.2 : A sample Physical Schema Graph

Now let us look at the transformation of a sample query. Suppose we have a physical schema graph as shown in Figure 4.2 and the following query:

Query Q2 :

```
        extract [ name, orientation, size,
                  extract [ name ] ( neighbors ),
                  extract [ name, price ] ( select [ name = ”computer” ] ( contains ) )
          ]
        ( RoomC )
```

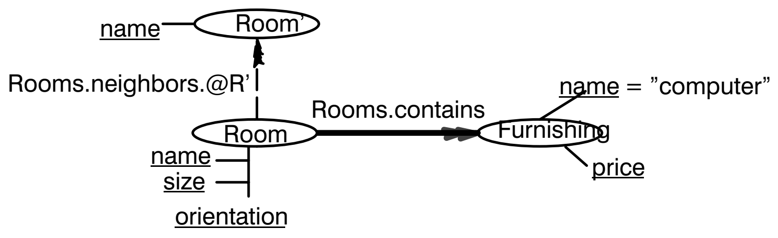

Figure 4.3: A sample Class Connection Graph

Figure 4.3 shows the Class Connection Graph constructed for that query. The nodes of the class connection graph represent classes which are affected by the given query. The edges are the functions involved in the query, connecting classes via implicit or explicit joins. There are three different kinds of edges, depending on the physical representation of these functions, i.e. did we store logical OIDs, physical references, or is this object function materialized. In our example, the *contains* function of *Room* objects is materialized, and the *neighbors* function is supported by pointers. The use of the *neighbors* function results in a second occurrence of class Room, here denoted as *Room’*.

In case a class extension would be stored vertically fragmented, this would result in having a class for each fragment, all of them connected with corresponding arcs. Further, we see the selection predicates (name = ”computer”) and the printable attributes, which shall be displayed (name, size, and orientation of Rooms).

A class connection graph represents a given query, without any hints on execution orders, that is, all possible execution plans can be generated from the class connection graph. For example, we have the choice to perform forward or backward traversals, or to start in the middle of a path, as well as to interleave other operations with the traversal.

4.2 Optimization of Query Execution Plans

In this section we describe the model of execution offered by the base system (DASDBS kernel and query processor) and how query execution plans are represented, before we describe the oportunities for optimization and how the optimization is perfomed.

To represent the execution plans, we use processing trees. Two possible processing trees are given in Figure 4.4. The leaf nodes, which represent complex operations on one DASDBS relation, are performed (in a set−oriented way) by the DASDBS kernel. Internal nodes are performed in the COOL−specific query processor (in a streaming mode).

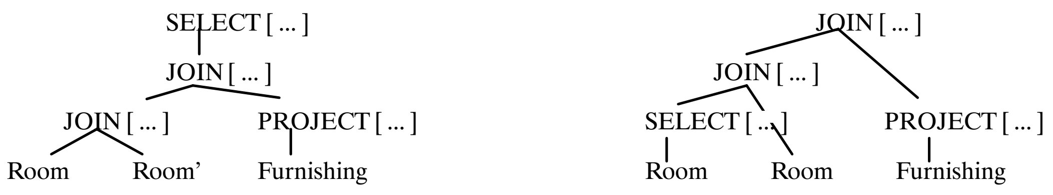

Figure 4.4: Two equivalent Processing Trees

Which Potential for Optimization do we have?

Given a query, obviously there are many equivalent processing trees, that is, alternatives to execute the query. These alternatives result from a number of open choices, some of which are listed in the following.

Join Ordering. A sequence of join operations is freely reorderable, this results in many alternatives. In order to reduce the number of orderings to consider, one may exclude the ones resulting in Cartesian products, or restrict the join orderings to the ones resulting in linear join trees, instead of considering all possible, bushy ones (this is the well−known problem studied, for example, in [Sel79,OL88].
The reason for join operations to occur is the following: First, reassembling objects from several relations requires a join operation for each partition. Object partitioning is introduced for example, in the mapping of inheritance hierarchies. Second, each application of an object valued function (which is not materialized) results in a corresponding join. Reordering these joins corresponds to changes to the order of function application, that is whether we do forward or backward traversals, or starting in the middle of a path query [Ber90]. Notice, that these implicit joins may be supported by link fields (see section 3.1), and therefore enable efficient pointer based join algorithms [SC90].

Pushing Selections. Selections may be performed at various times. For example, one could perform a given selection before or after applying an object valued function (move selection into join, or vice versa). Additionally, selections with conjunctively combined selection predicates may be split into two selections (or vice versa), and the order of selections, as well as the order of the terms in selection predicates, may be changed.

Pushing Projections. Projections, in the same way as selections, may be performed at various times. In addition, the order of applying projections and selections may be changed.

Method Selection. For each logical operation there may be several methods, that is physical implementations. For example, a join operation may be performed by nested loops join, hash join, or sort merge join. In addition, for implicit joins which are supported by link fields, pointer based join versions may be selected. Further, selections may be performed by using indexes, or by performing a relation scan, followed by predicate testing.

Index Selection. If a selection is supported by multiple indices, one has the opportunity to use all, some, or just one of the available indices.

Almost all of the choices given above, can be combined orthogonally, such that the number of equivalent processing trees grows extremely fast with increasing query complexity.

How to find the Optimal Execution Plan

Due to the fact that a query optimizer is one of the most intricate subsystems of a database system, it is desirable not to start an implementation from scratch. Therefore, we decided to use the EXODUS optimizer generator for the implementation of the COOL optimizer. Without going into detail, a cost model to estimate the quality of execution plans, as well as rules to describe possible transformations have to be defined by the implementor of the optimizer, the search strategy is provided by EXODUS. The cost functions for methods are fairly standard, similar to the ones of System R [Sel79]. Rules describing possible transformations of processing trees, that is, of nested relational algebra expressions, are derived from transformation rules known from the (flat) relational model. Roughly, 1NF rules are applicable in the NF^2 model on relations and on the subrelations, having additional constraints and more complex argument transfer functions. Further, additional (complex) rules have been defined for the nested model [Sch86, Sch88].

A first prototype optimizer has been implemented, and integrated in the COCOON−DASDBS system. COOL queries are passed from the COOL interface to the optimizer, and after translation onto the physical schema and optimization, execution plans can be passed to the query processor to execute the query on the DASDBS kernel. The functionality includes removal of redundant operations, the combination of operations, and select−project−join ordering, for nested relations. Nested Loop−, Nested Block−, and Sort Merge Joins are considered as implementation strategies. After the completion of the implementations of pointer based join algorithms, these methods will also be added to the optimizer's repertoire. Since our goal was to evaluate the advantages and cost of using a complex record (DASDBS) instead of flat record (RDBMS) storage manager, the main emphasis has been on the effects of hierarchical clustering and embedded references. Indexes played only a supporting role; future improvements will include indexes as well.

The implementation of COOL on top of DASDBS has been completed (to the described level) only recently. Therefore, we can not yet provide any performance Figures. The next step in our work, however, is to run such experiments and compare the relative performance of several possible physical designs. Further, the DASDBS implementation will also be evaluated in comparison with to other implementations based on commercial platforms: Oracle, as a relational system that can emulate two−level nesting (via Oracle "Clusters"), and Ontos, as one of the early commercial OODBMSs. First experiences with these two implementations are reported in [TS91].

References

[Ber90] Bertino, E., "Optimization of Queries Using Nested Indices," Lecture Notes in Computer Science, vol. 416, p. 44, Springer Verlag, Venice, Italy, March 1990.

[GD87] Graefe, G. and DeWitt, D.J., "The EXODUS Optimizer Generator," Proc. ACM SIGMOD Conf., p. 160, San Francisco, CA, May 1987.

[KM90a] Kemper, A. and Moerkotte, G., "Advanced Query Processing in Object Bases Using Access Support Relations," Proc. Int'l. Conf. on Very Large Data Bases, p. 290, Brisbane, Australia, 1990.

[KM90b] Kemper, A. and Moerkotte, G., "Access Support in Object Bases," Proc. ACM SIGMOD Conf., p. 364, Atlantic City, NJ, May 1990.

LV91] Laanzelotte, R., Valduriez, P., Ziane, M., and Cheiney, J.J., "Optimization of Nonrecursive Queries in OODBs," Proc. Conf. on Deductive and Object−oriented Databases, Munich, Germany, December 1991.

[OL88] Ono, K. and Lohman, G.M., "Extensible Enumeration of Feasible Joins for Relational Query Optimization," IBM Research Report, vol. RJ 6625 (63936), San Jose, CA, December 1988.

[Sch86] Scholl, M., "Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations," Proc. ICDT, p. 380, Rome, Italy, September 1986

[Sch88] Scholl, M., The Nested Relational Model − Efficient Support for a Relational Database Interface, Dissertation, TH Darmstadt, 1988

[Sel79] Selinger, P. Griffiths, Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G., "Access Path Selection in a Relation−al Database Management System," Proc. ACM SIGMOD Conf., p. 23, Boston, MA, May−June 1979. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan−Kaufman, San Mateo, CA, 1988

[SC90] Shekita, E.J. and Carey, M.J., "A Performance Evaluation of Pointer−Based Joins," Proc. ACM SIGMOD Conf., p. 300, Atlantic City, NJ, May 1990.

[SS90] Scholl, M., Schek, H.−J., A Relational Object Model, in Abiteboul, S. and Kanallaiks, P.C., editors, ICDT'90 − Proc. Int'l. Conf. on Database Theory, pages 89−105, Paris, December 1990, LNCS 470, Springer Verlag, Heidelberg

[TS91] Tresch, M., Scholl, M., Implementing an Object Model on Top of Commercial Database Systems, in Scholl, M., editor, Proc. 3rd GI Workshop on Foundations of Database Systems, Volkse, Germany, Technical Report 158, Dept. of Computer Science, ETH Zurich, May 1991