Query Processing in Object-Oriented Database Systems

M. Tamer Özsu

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2H1
ozsu@cs.ualberta.ca

José A. Blakeley

Texas Instruments Incorporated
P.O. Box 655474, MS 238
Dallas, Texas 75265
USA
blakeley@csc.ti.com

Abstract

One of the basic functionalities of database management systems (DBMSs) is to be able to process declarative user queries. The first generation of object-oriented DBMSs did not provide declarative query capabilities. However, the last decade has seen significant research in defining query models (including calculi, algebra and user languages) and in techniques for processing and optimizing them. Many of the current commercial systems provide at least rudimentary query capabilities. In this chapter we discuss the techniques that have been developed for processing object-oriented queries. Our particular emphasis is on extensible query processing architectures and techniques. The other chapters in this book on query languages and optimization techniques complement this chapter.

1 Introduction

One of the criticisms of first-generation object-oriented database management systems (OODBMSs) was their lack of declarative query capabilities. This led some researchers to brand first generation (network and hierarchical) DBMSs as object-oriented [Ullman 1988]. It was commonly believed that the application domains that OODBMS technology targets do not need querying capabilities. This belief no longer holds, and declarative query capability is accepted as one of the fundamental features of OODBMSs [Atkinson et al. 1989; Stonebraker et al. 1990]. Indeed, most of the current prototype systems experiment with powerful query languages and investigate their optimization. Commercial products have started to include such languages as well (e.g., O_2 [Deux et al. 1991], ObjectStore [Lamb et al. 1991]).

In this chapter we discuss the issues related to the *optimization* and *execution* of OODBMS query languages (which we collectively call *query processing*). Query optimization techniques are dependent upon the query model and language. For example, a functional query language lends itself to functional optimization which is quite different from the algebraic, cost-based optimization techniques employed in relational as well as a number of object-oriented systems. The query model, in turn, is based on the data (or object) model since the latter defines the access primitives which are used by the query model. These primitives, at least partially, determine the power of the query model. Despite this close relationship, in this chapter we do not consider issues related to the design of object models, query models, or query languages in any detail. Language design issues are discussed elsewhere in this book. The interrelationship between object and query models is discussed in [Blakeley 1991; Özsu and Straube 1991; Özsu et al. 1993; Yu and Osborn 1991].

Almost all object query processors proposed to date use optimization techniques developed for relational systems. However, there are a number of issues that make query processing more difficult in OODBMSs. The following are some of the more important issues:

1. **Type system.** Relational query languages operate on a simple type system consisting of a single aggregate type: *relation*. The closure property of relational languages implies that each relational operator takes one or more relations as operands and produces a relation as a result. In contrast, object systems have richer type

systems. The results of object algebra operators are usually sets of objects (or collections) whose members may be of different types. If the object languages are closed under the algebra operators, these heterogeneous sets of objects can be operands to other operators. This requires the development of elaborate type inferencing schemes to determine which methods can be applied to **all** the objects in such a set. Furthermore, object algebras often operate on semantically different collection types (e.g., set, bag, list) which imposes additional requirements on the type inferencing schemes to determine the type of the results of operations on collections of different types.

2. **Encapsulation.** Relational query optimization depends on knowledge of the physical storage of data (access paths) which is readily available to the query optimizer. The encapsulation of methods with the data that they operate on in OODBMSs raises (at least) two issues. First, estimating the cost of executing methods is considerably more difficult than estimating the cost of accessing an attribute according to an access path. In fact, optimizers have to worry about optimizing method execution, which is not an easy problem because methods may be written using a general-purpose programming language. Second, encapsulation raises issues related to the accessibility of storage information by the query optimizer. Some systems overcome this difficulty by treating the query optimizer as a special application that can break encapsulation and access information directly [Cluet and Delobel 1992]. Others propose a mechanism whereby objects "reveal" their costs as part of their interface [Graefe and Maier 1988].

3. **Complex objects and inheritance.** Objects usually have complex structures where the state of an object references other objects. Accessing such complex objects involves *path expressions*. The optimization of path expressions is a difficult and central issue in object query languages. We discuss this issue in some detail in this chapter. Furthermore, objects belong to types related through inheritance hierarchies. Efficient access to objects through their inheritance hierarchies is another problem that distinguishes object-oriented from relational query processing.

4. **Object models.** OODBMSs lack a universally accepted object model definition. Even though there is some consensus on the basic features that need to be supported by any object model (e.g., object identity, encapsulation of state and behavior, type inheritance, and typed collections), how these features are supported differs among models and systems. As a result, the numerous projects that experiment with object query processing follow quite different paths and are, to a certain degree, incompatible, making it difficult to amortize on the experiences of others. This diversity of approaches is likely to prevail for some time, therefore, it is important to develop extensible approaches to query processing that allow experimentation with new ideas as they evolve. We provide an overview of various extensible object query processing approaches.

The organization of this chapter is as follows. In Section 2 we present several representative query processing architectures that have been developed and experimented with. These architectures, in one sense, define the query processing methodology. In Section 3, we discuss the various approaches to optimization of object queries. Section 4 is devoted to a discussion of query execution strategies. We conclude, in Section 5, with some comments on the state-of-the-art in object query processing and optimization and the work that remains to be done.

2 Query Processing Architecture

In this section we focus on two architectural issues: the query processing methodology and the query optimizer architecture.

2.1 Query Processing Methodology

A query processing methodology similar to relational DBMSs, but modified to deal with the difficulties discussed in the previous section, can be followed in OODBMSs. Figure 1 depicts such a methodology proposed in [Straube and Özsu 1990a].

The steps of the methodology are as follows. Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first

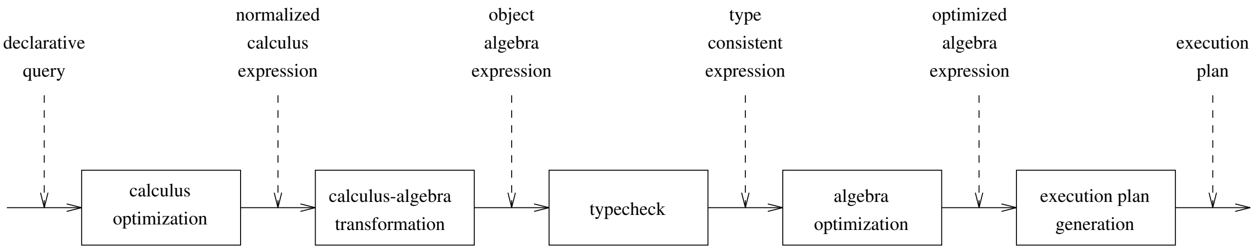

Figure 1: Object query processing methodology

reduced to a normalized form by eliminating duplicate predicates, applying identities and rewriting. The normalized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent extents of classes in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested functions. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence-preserving rewrite rules [Freytag 1987] to the type consistent algebra expression. Lastly, an execution plan which takes into account object implementations is generated from the optimized algebra expression. The separation of algebraic optimization step from the execution plan generation step follows the distinction that is made between "query rewrite" and "plan optimization" [Haas et al. 1989]. Query rewrite is a high-level process where general-purpose heuristics drive the application of transformation rules. Plan optimization, on the other hand, is a lower level process which transforms a query into the most cost effective access plan, based on a specific cost model and knowledge of access paths and database statistics. This methodology clearly separates the various concerns and provides extensibility to the query processor. However, it faces one serious problem: the combinatorial cost of analyzing the large number of plans that are generated. The algebraic optimization step generates a family of equivalent query expressions based on the transformation rules defined for the algebra. The execution plan generation step generates a number of alternative mappings from each of these expressions to the object manager interface calls. Therefore, the number of alternatives that need to be considered may become quite high. One alternative followed in Starburst [Haas et al. 1989] is to use heuristic rules to control query rewrite so that a single query expression is generated as input to the plan optimization step. Cost-based optimization approaches, on the other hand, merge these two steps into one and consider the alternative execution algorithms as part of the search space.

This methodology assumes the existence of a fully specified calculus-based language and an object algebra. There are only a few calculi that have been defined for OODBMSs [Abiteboul and Beeri 1987; Peters et al. 1993; Straube and Özsu 1990] and a few object logics with declarative query facilities [Kifer and Wu 1989; Maier 1986]. There are a large number of declarative user languages (e.g., [Blakeley 1991; Carey et al. 1988; Kifer et al. 1992; Orenstein et al. 1992]), but these generally do not have a formal calculi. Some of these languages are discussed elsewhere in this book. Work on object algebras has been more prevalent. There are many algebras that have been defined with varying operations (e.g., [Alhajj 1993; Beeri and Kornatzky 1990; Blakeley et al. 1993; Peters et al. 1993; Shaw and Zdonik 1990; Straube and Özsu 1990a; Vandenberg and DeWitt 1991]).

2.2 Optimizer Architecture

Query optimization can be modeled as an optimization problem whose solution is the choice of the "optimum" *state* in a *state space* (also called *search space*). In query optimization, each state corresponds to an algebraic query indicating an execution schedule and represented as a processing tree. The state space is a family of equivalent (in the sense of generating the same result) algebraic queries. Query optimizers generate and search a state space using a *search strategy* applying a *cost function* to each state and finding one with minimal cost[1]. Thus, to characterize a query optimizer three things need to be specified:

[1] In this chapter we are mostly concerned with cost-based optimization, which is arguably the more interesting case.

1. the search space and the the transformation rules that generate the alternative query expressions which constitute the search space;

2. a search algorithm that allows one to move from one state to another in the search space; and

3. the cost function that is applied to each state.

Many existing OODBMS optimizers are either implemented as part of the object manager on top of a storage system, or they are implemented as client modules in a client-server architecture. In most cases, the above mentioned four aspects are "hardwired" into the query optimizer. Given that extensibility is a major goal of OODBMSs, one would hope to develop an extensible optimizer that accommodates different search strategies, different algebra specifications with their different transformation rules, and different cost functions. Rule-based query optimizers [Freytag 1987; Graefe and DeWitt 1987] provide a limited amount of extensibility by allowing the definition of new transformation rules. However, they do not allow extensibility in other dimensions. In this section we discuss some new promising proposals for extensibility in OODBMSs.

The Open OODB project [Wells et al. 1992] at Texas Instruments[2] concentrates on the definition of an open architectural framework for OODBMSs and on the description of the design space for these systems. Query processing in Open OODB [Blakeley et al. 1993] follows steps similar to Figure 1. The query module is an example of intra-module extensibility in Open OODB. The query optimizer, built using the Volcano optimizer generator [Graefe and McKenna 1993], is extensible with respect to algebraic operators, logical transformation rules, execution algorithms, implementation rules (i.e., logical operator to execution algorithm mappings), cost estimation functions, and physical property enforcement functions (e.g., presence of objects in memory). The clean separation between the user query language parsing structures and the operator graph on which the optimizer operates allows the replacement of the user language or optimizer. The separation between algebraic operators and execution algorithms allows exploration with alternative methods for implementing algebraic operators. Code generation is also a well defined subcomponent of the query module which facilitates porting the query module to work on top of other OODBMSs. The Open OODB query processor includes a query execution engine containing efficient implementations of scan, indexed scan, hybrid-hash join [Shapiro 1986], and complex object assembly [Keller et al. 1991].

The EPOQ project [Mitchell et al. 1993] is another approach to query optimization extensibility, where the search space is divided into *regions*. Each region corresponds to an equivalent family of query expressions that are reachable from each other. The regions are not necessarily mutually exclusive (Figure 2) and differ in the queries that they manipulate, control (search) strategy that they use, query transformation rules that they incorporate, and optimization objectives they achieve. For example, one region may cover transformation rules that deal with simple select queries, while another region may deal with transformations for nested queries. Similarly, one region may have the objective of minimizing a cost function, while another region may attempt to transform queries in some desirable form. Each region may be nested to a number of levels, allowing hierarchical search within a region. Since the regions do not represent equivalence classes, there is a need for a global control strategy to determine how the query optimizer moves from one region to another. The feasibility and effectiveness of this approach remains to be verified.

The TIGUKAT project [Peters et al. 1992] uses an object-oriented approach to query processing extensibility. TIGUKAT object model is an extensible uniform behavioral model characterized by a purely behavioral semantics and a uniform approach to objects. The model is behavioral in that the only way objects are accessed is by applying behaviors (which replace both the instance variables and the methods available in other object models) to objects. Behaviors are defined on types and their implementations are modeled as functions. Every concept, including types, classes, collections, meta-information, etc. is a first-class object. The uniformity of the object model extends to the query model, treating queries as first-class objects [Peters et al. 1993]. A `Query` type is defined as a subtype of the `Function` type (Figure 3). Thus, queries are specialized kinds of functions that can be compiled and executed. Furthermore, `Query` type can be specialized based on a classification scheme. Figure 3 shows, for example, specialization of ad hoc and production queries. The inputs and outputs of queries are collections (which are also objects), providing closure.

TIGUKAT query optimizer [Muñoz 1994] follows the same philosophy of representing system concepts as objects and is along the lines of [Lanzelotte and Valduriez 1991]. The search space, the search strategy and the cost function

[2] This is different than HP's Open ODB product.

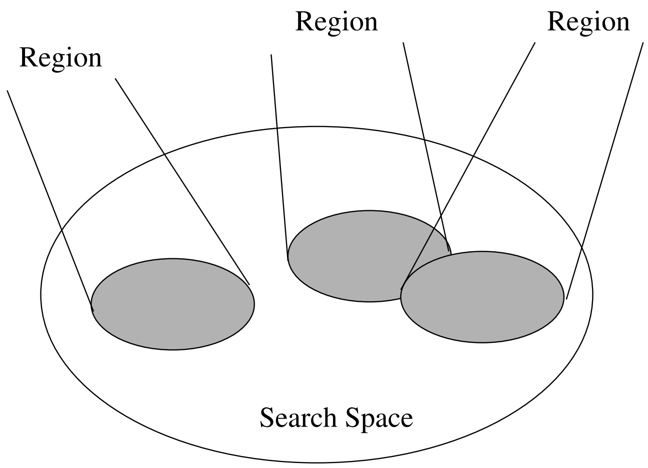

Figure 2: Partitioning the search space into regions

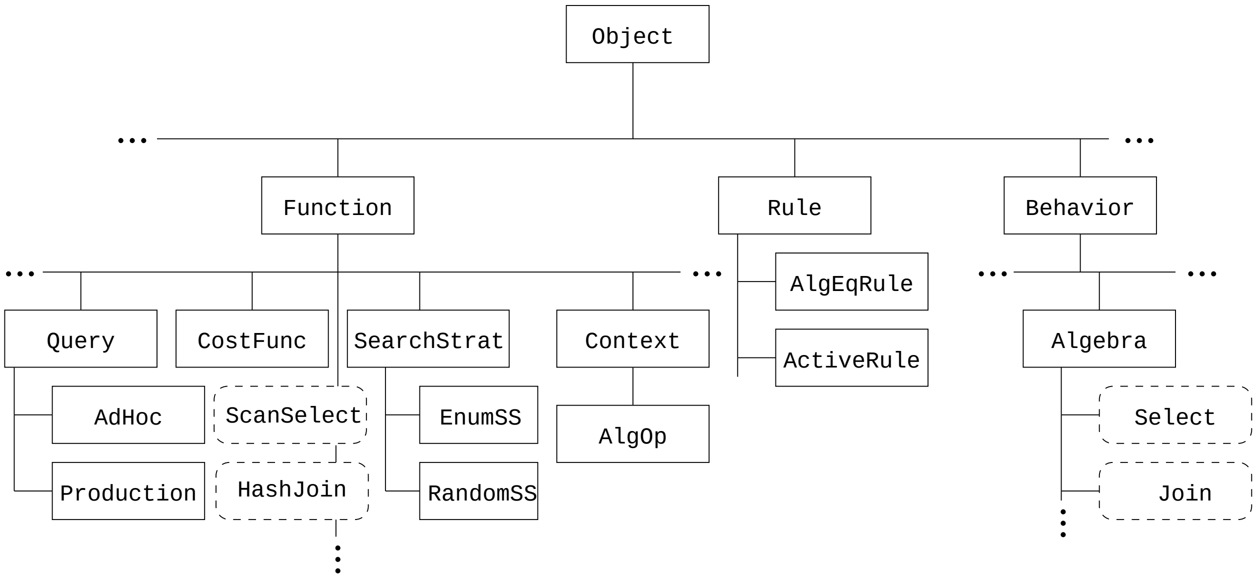

Figure 3: Optimizer as part of the type system

are modeled as objects (see Figure 3). The incorporation of these components of the optimizer into the type system provide extensibility via the basic object-oriented principle of subtyping and specialization.

The states in the search space are modeled as processing trees (PT) whose leaf nodes are references to collections and non-leaf nodes denote behavior applications whose results are other objects. Those nodes which correspond to algebraic operator behaviors return temporary collections as result.

Algebraic operators (e.g., `Select`, `Join`) are defined as behaviors of the `Collection` type. They are modeled as instances (shown as dashed boxes in Figure 3) of type `Algebra` which is a subtype of type `Behavior`. The implementation (execution) algorithms for these algebraic operators are modeled as function objects (e.g., `HashJoin`, `ScanSelect`). These implementation functions cannot be used as nodes of a PT, since these nodes should represent execution functions all of whose arguments have been marshalled. Therefore, `AlgOp` type is defined whose instances are functions with marshalled arguments and represent nodes of a PT. In this fashion, each node of a PT represents a specific execution algorithm for an algebra expression.

Search strategies are similarly modeled as objects, but separate from the search space. `SearchStrat` is defined as a subtype of type `Function`, and it can in turn be specialized. Figure 3 shows the specialization of `SearchStrat` into enumerated search strategies `EnumSS` and randomized search strategies `RandomSS`. The algebraic transformation rules that control the movement of the search strategy through the search space are implemented as instances of `AlgEqRule` which is a subtype of `Rule`.

Cost functions (instances of `CostFunc`) are defined as special types of functions, making them first-class objects. Each function is associated a cost through the behavior *B_costFunction*. Application of this behavior to a function object f (i.e., $f.B\_costFunction$) returns another function object g of type `CostFunc` that implements the computation of the cost of executing function f. This allows definition of parameterized cost functions whose values are dependent upon a number of factors.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer the extensibility inherent in object models. The optimizer basically implements a control strategy that associates a search strategy and a cost function to each query.

3 Optimization Techniques

Our discussion of optimization techniques for object queries follows two fundamental directions. The first is the cost-based optimization of queries based on algebraic manipulations. Algebraic optimization techniques have been studied quite extensively within the context of the relational model. The work on relational DBMSs has benefited greatly from the availability of a universally accepted algebra definition. Despite over two dozen proposals, there is no universally accepted object algebra, making it very difficult to generalize research results. The second is the optimization of *path expressions*. Path expressions represent traversal paths between objects and are unique to OODBMSs. The optimization of path expressions is an important issue in OODBMSs and has been a subject of considerable investigation.

3.1 Algebraic Optimization

We previously identified the components of an algebraic optimizer. In this section, we discuss each of these components in some detail.

Search Space and Transformation Rules

A major advantage of algebraic optimization is that an algebraic query expression can be transformed using well-defined algebraic properties such as transitivity, commutativity and distributivity. Therefore, each query has a (potentially large) number of equivalent expressions, which make up the *search space*. These expressions are equivalent in terms of the results that they generate, but may be widely different in terms of their costs. Thus, the query optimizers modify the query expressions, by means of algebraic transformation rules, in an attempt to obtain one which generates the same result with the lowest possible cost.

The transformation rules are very much dependent upon the specific object algebra, since they are defined individually for each object algebra and for their combinations. The lack of a standard object algebra definition is particularly troubling since the community cannot benefit from generalizations of numerous object algebra studies. The general considerations for the definition of transformation rules and the manipulation of query expressions is quite similar to relational systems, with one particularly important difference. Relational query expressions are defined on flat relations, whereas object queries are defined on classes (or collections or sets of objects) that have inheritance relationships among them. It is, therefore, possible to use the semantics of these relationships in object-oriented query optimizers to achieve some additional transformations.

Consider, for example, the object algebra defined in [Straube and Özsu 1990a] whose operators work on and produce sets of objects. Consider three operators `union` (denoted $\cup$), `intersection` (denoted $\cap$) and parameterized `select` (denoted $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$), where `union` and `intersection` have the usual set-theoretic semantics and `select` selects objects from one set P using the sets of objects $Q_1 \ldots Q_k$ as parameters (in a sense a generalied form of semijoin). The following are some of the transformation rules that can be applied during optimization to get equivalent query expressions (for brevity we use $QSet$ to denote $Q_1 \ldots Q_k$; $RSet$ is defined similarly):

$$(P\ \sigma_{F_1}\ \langle QSet\rangle)\ \sigma_{F_2}\ \langle RSet\rangle \quad\Leftrightarrow\quad (P\ \sigma_{F_2}\ \langle RSet\rangle)\ \sigma_{F_1}\ \langle QSet\rangle \tag{1}$$

$$(P \cup Q)\ \sigma_F\ \langle RSet\rangle \quad\Leftrightarrow\quad (P\ \sigma_F\ \langle RSet\rangle) \cup (Q\ \sigma_F\ \langle RSet\rangle) \tag{2}$$

$$(P\ \sigma_{F_1}\ \langle QSet\rangle)\ \sigma_{F_2}\ \langle RSet\rangle \quad\Leftrightarrow\quad (P\ \sigma_{F_1}\ \langle QSet\rangle)\ \cap\ (P\ \sigma_{F_2}\ \langle RSet\rangle) \tag{3}$$

Rule 1 captures commutativity of `select`, while rule 2 denotes that `select` distributes over `union`. Rule 3 is an identity which utilizes the fact that `select` merely restricts its input and returns a subset of its first argument[3].

The first two rules are quite general in that they represent equivalences that are inherited from set theory. The third one is a special transformation rule for a specific object algebra operator defined with a specific semantics. All three, however, are syntactic in nature. Consider the following rules, on the other hand, where C_i denotes the set of objects in the extent of class c_i and C_j^* denotes the deep extent of class c_j (i.e., the set of objects in the extent of c_j as well as in the extents of all those which are subclasses of c_j):

$$C_1 \cap C_2 \quad = \quad \phi \ \text{ if } c_1 \neq c_2 \tag{4}$$

$$C_1 \cup C_2^* \quad = \quad C_2^* \ \text{ if } c_1 \text{ is a subclass of } c_2 \tag{5}$$

$$\begin{aligned}(P\ \sigma_F\ \langle QSet\rangle) \cap R \quad &\overset{c}{\Leftrightarrow} \quad (P\ \sigma_F\ \langle QSet\rangle) \cap (R\ \sigma_{F'}\ \langle QSet\rangle)\\ &\overset{c}{\Leftrightarrow} \quad P \cap (R\ \sigma_{F'}\ \langle QSet\rangle)\end{aligned} \tag{6}$$

These transformation rules are semantic in nature since they depend on the object model and query model specifications. Rule 4, for example, is true because the object model restricts each object to belong to only one class. Rule 5 holds because the query model permits retrieval of objects in the deep extent of the target class. Finally, rule 6 relies on type consistency rules [Straube and Özsu 1990b] for its applicability as well as the condition that F' is identical to F except each occurrence of p is replaced by r (this last condition is denoted by the c over the $\Leftrightarrow$).

Since the idea of query transformation is well-known, we do not elaborate on the techniques. The above discussion demonstrates the general idea and also highlights the unique aspects that need to be considered in object algebras. In Section 3.2, we discuss the transformations for a new operator called `materialize` recently proposed to optimize path expressions.

Search Algorithm

Exhaustive search, whereby the entire search space is enumerated, is the most straight-forward search strategy that can be used. A cost function can be applied to each equivalent expression to determine the cheapest. An improvement is to use a dynamic programming approach whereby new expressions are constructed bottom-up using the previously determined optimal subexpressions [Lee et al. 1988; Selinger et al. 1979]. The Volcano optimizer generator uses a top-down, dynamic programming approach to search with branch-and-bound pruning [Graefe and McKenna 1993]. These are what we call *enumerative algorithms*.

Since the enumerative search algorithms are based on evaluating the cost of the entire search space, their overhead is quite high. In relational systems, the number of join operations typically determine the complexity of enumerative search. If there are N join operations and there are two choices for join ordering (for inner and outer relations), then there are $O(2^N)$ alternative query expressions to evaluate[4]. Heuristics such as performing selections and projections before joins (to reduce the sizes of the join operands) do not change the combinatorial nature of the problem. Therefore, the value of N and the threshold beyond which combinatorial nature of the problem makes enumerative solutions infeasible becomes an important issue. $N = 10$ has been suggested as an empirical threshold value [Ioannidis and Wong 1987].

[3]These rules make assumptions about the formulae (F_i) which we do not get into in this chapter.

[4]More accurate bounds for various types of join queries (linear and star) are given in [Ono and Lohman 1990]. However, the combinatorial nature of the problem remains.

This may be an important concern for object query optimization as well. First, a large number of the object algebras have join operators or operators with semantics similar to join. Second, it has been suggested [Ioannidis and Wong 1987] that even if it is not common to find many business data processing queries with more than 10 join operations, such queries are quite common in AI and decision support system applications. These are important applications that OODBMSs attempt to serve. Finally, as we will address in Section 3.2, one method of executing path expressions is to represent them as explicit joins and then use the well-known join algorithms to optimize them. If this is the case, then the number of joins and other operations with join semantics in a query is quite likely to be higher than this empirical threshold of 10.

In these cases, *randomized search algorithms* have been suggested as one alternative to restrict the region of the search space that is analyzed. Randomized algorithms are well-known in operations research and two versions of these algorithms have been investigated within the context of relational query optimization: *simulated annealing* [Ioannidis and Wong 1987] and *iterative improvement* [Swami 1989]. A combination of the two algorithms, called *two-phase optimization*, is proposed in [Ioannidis and Cha Kang 1990]. Without getting into the details of these algorithms, the general idea can be described as follows. The randomized algorithms start from a random state in the search space (i.e., an initial query expression which may be obtained as a result of query translation) and then "walk" through the search space, evaluating the cost of each state and stopping either when they estimate that they have found the optimum execution plan or when a predetermined optimization time expires. The walking between states is controlled by the transformation rules such as the ones described in the previous section and a global control strategy. Iterative improvement accepts a move from one state to another only if the cost of the destination state is lower than the cost of the source state. Simulated annealing, on the other hand, allows a move to a higher-cost state with a certain probability which diminishes as optimization time moves along.

Since these are heuristic algorithms investigating only a portion of the search space, they cannot be guaranteed to be optimal. However, it has been shown that the randomized techniques converge to a state which is fairly close to the optimal state given sufficient time.

There has not been any study of randomized search algorithms within the context of OODBMSs. The general strategies are not likely to change, but the tuning of the parameters and the definition of the space of acceptable solutions should be expected to change. It is also interesting to note the surface similarity between randomized search algorithms and the regions approach proposed by Mitchell, et al [1993]. Further studies are required to establish the relationship more firmly.

Cost Function

Typical cost functions used in query optimization take into account the various costs that are incurred in processing the query. In non-distributed systems, this is typically the I/O and CPU cost, while in distributed systems, the communication cost is also added.

The arguments to cost functions are based on various information regarding the storage of the data. Typically, the optimizer considers the number of data items (cardinality), the size of each data item, its organization (e.g., whether there are indexes on it or not) and others. This information is readily available to the query optimizer in relational systems (through the system catalog), but may not be in OODBMSs. As indicated in the introduction, there is a controversy in the research community as to whether the query optimizer should be able to break the encapsulation of objects and look at the data structures used to implement them. If this is permitted, then the cost functions can be specified similar to relational systems [Blakeley et al. 1993; Cluet and Delobel 1992; Dogac et al. 1994; Orenstein et al. 1992]. Otherwise, an alternative specification has to be considered.

The cost function can be defined recursively based on the algebraic processing tree. If the internal structure of objects is not visible to the query optimizer, the cost of each node (representing an algebraic operation) has to be defined. One way to define it is to have objects "reveal" their costs as part of their interface [Graefe and Maier 1988]. A similar approach is one that is provided in the TIGUKAT project [Muñoz 1994]. Since the algebraic operations are behaviors defined on type `Collection`, the nodes of the algebraic processing tree are behavior applications. There may be various functions that implement each behavior (representing different execution algorithms), in which case the behaviors "reveal" their costs as a function of (a) the execution algorithm, and (b) the collection over which they operate. The bottom line, in both cases, is the same: let the type implementer specify a more abstract cost function for behaviors from which the query optimizer can calculate the cost of the entire processing tree. The definition of cost functions, especially in the approaches based on the objects revealing their costs, needs to be investigated further before satisfactory conclusions can be reached.

Parameterization

Compilation-time query optimization is a static process in the sense that the optimizer makes use of the database statistics at the time the query is compiled and optimized in selecting the optimal execution plan. This decision is independent of the execution-time statistics such as the system load. Further it does not take into account the changes to the database statistics as a result of updates that may occur between the time the query is optimized and the time it is executed. This is especially a problem in production-type queries which are optimized once (with considerable overhead) and executed a large number of times. This may be an even more serious issue in OODBMSs which may be used as repositories for design prototypes (software or otherwise). These databases are by definition more volatile, resulting in significant changes to the database (that is why dynamic schema evolution is so important in OODBMSs). The query optimization strategy has to be able to cope with these changes.

This issue can be handled in one of two ways. One alternative is to determine an optimization/re-optimization interval and re-optimize the query periodically. Even though this is a simple approach, it is based on a fixed time interval whose determination in general would be problematic. A slight variation may be to determine the re-optimization point based on the difference between the actual execution time and the estimated execution time. Consequently, the run-time system can track the actual execution time and whenever it deviates from the estimated time by more than a fixed threshold, the query is re-optimized. Again, the determination of this threshold would be a concern as well as the run-time overhead of tracking query execution.

An alternative that has been researched [Graefe and Ward 1989; Ioannidis et al. 1992] and implemented in ObjectStore [Orenstein et al. 1992] is *parametric query optimization*, which is also called *dynamic plan selection*. In this case, the optimizer maintains multiple execution strategies at compile time and makes a final plan selection at run-time based on various system parameters and the current database statistics. This approach may also fit well with the methodology depicted in Figure 1 using an optimizer that respects the encapsulation of objects. In this case, algebraic optimization can ignore all physical execution characteristics, instead generating a set of "desirable" (however defined) equivalent query expressions which are handed over to the object manager which is responsible for storing objects. The object manager can then compare the alternatives (at run time) based on their execution characteristics. However, this approach also has the significant problem of potentially incurring high run-time overhead.

A problem with compile-time parametric optimization (and run-time resolution) is the potential exponential explosion of the dynamic plans as a function of the complexity of the query and the number of optimization parameters unknown at compile time. This problem, along with the problems of error propagation and inaccuracy of selectivity and cost estimation methods, makes "run-time" query optimization an attractive alternative.

3.2 Path Expressions

Most query languages allow queries whose predicates involve conditions on object access along reference chains. These reference chains are often called *path expressions* [Zaniolo 1983]; in the past they have also been called *complex predicates* or *implicit joins* [Kim 1989]. Optimizing the computation of path expressions is a problem that has received substantial attention in object-query processing.

Path expressions allow a succinct, high-level notation for expressing navigation through the *object composition graph* which enables the formulation of predicates on values deeply nested in the structure of an object. Path expressions provide a uniform mechanism for the formulation of queries that involve object composition and inherited member functions. Path expressions may be *single-valued* or *set-valued*, and may appear in a query as part of a predicate, as a target to a query (when set-valued), or as part of a projection list. Techniques to traverse path expressions forward and backward are presented by [Jenq et al. 1990].

Let $p.m_1(\alpha_1).m_2(\alpha_2).\cdots.m_n(\alpha_n)$ be a path expression, where p is a variable representing an object instance; α_i, $1 \leq i \leq n$, represents a (possibly empty) list of arguments to the corresponding function m_i. If all m_i are single-valued functions, then we call the path expression *single-valued.* If at least one of the m_i is set-valued, then the path expression is *set-valued.*

The problem of optimizing path expressions spans the entire query-compilation process. During or after parsing of a user query but before algebraic optimization, the query compiler must recognize what path expressions can potentially be optimized. This is typically achieved through *rewriting* techniques which transform path expressions into equivalent logical algebra expressions [Cluet and Delobel 1992]. Once path expressions are represented in algebraic form, the query optimizer explores the space of *equivalent algebraic* and execution plans searching for one of minimal cost [Lanzelotte and Valduriez 1991; Blakeley et al. 1993]. Finally, the optimal execution plan may involve algorithms to efficiently compute path expressions including hash-join [Shapiro 1986], complex-object assembly [Keller et al. 1991], or indexed scan through path indexes [Maier and Stein 1986; Valduriez 1987; Kemper and Moerkotte 1994].

To illustrate this point, consider the query to retrieve all employee names, their departments, and job descriptions for employees working in a department located in a Dallas plant. We will use the OQL[C++] object query-language syntax as an example user query language [Blakeley 1994]. Figures 4 illustrates several key stages in processing this query.

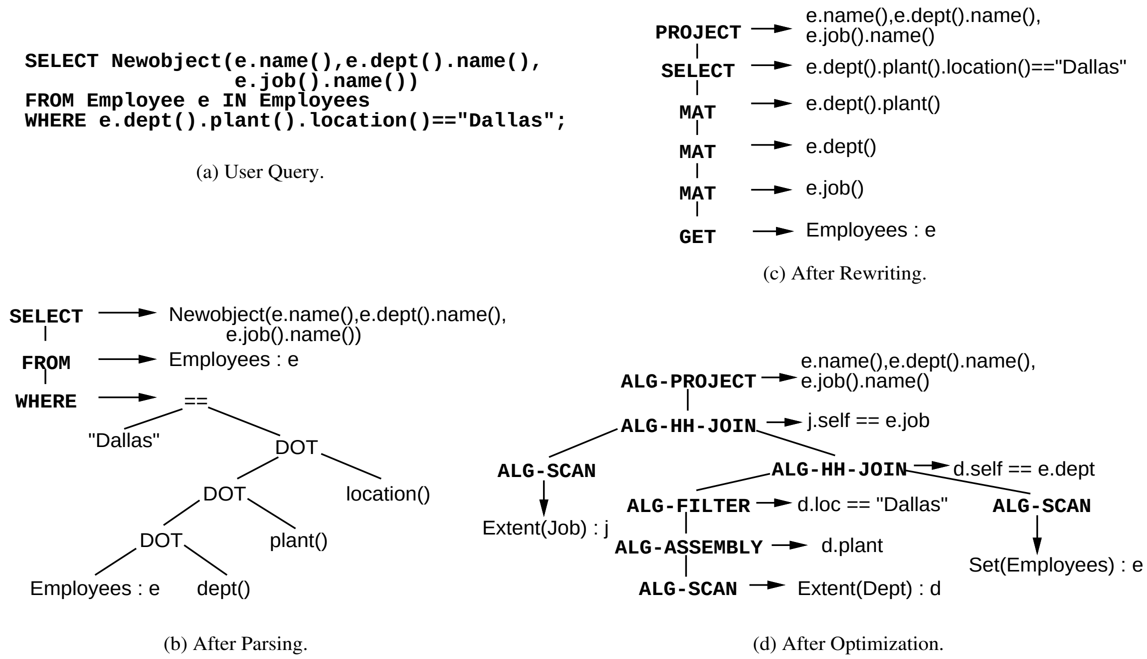

Figure 4: Optimization of path expressions.

Rewriting

Consider the path expression e.dept().plant().location(). Assume every employee instance has a reference to a department, each department has a reference to a plant, and each plant instance has a location field. Also, assume that department and plant types have a corresponding type extent. The first two links of the above path may involve the retrieval of department and plant objects from disk. The third path involves only a lookup of a field within a plant object. Therefore, only the first two links present opportunities for query optimization in the computation of that path. An object-query compiler needs a mechanism to distinguish these links in a path representing possible optimizations.

This is typically achieved through a *rewriting* phase.

Cluet and Delobel [1992] describe a type-based rewriting technique to be used as a basis for a new query optimizer being implemented in O_2 [Deux et al. 1991]. Their approach "unifies" algebraic and type-based rewriting techniques, permits factorization of common subexpressions, and supports heuristics to limit rewriting. They exploit type information to decompose initial complex arguments of a query into a set of simpler operators and to rewrite path expressions ("pointer chasing") into joins. Lanzelotte and Valduriez [1991] present a similar attempt to optimizing path expressions within an algebraic framework using an operator called `implicit join`.

Algebraic optimization

Blakeley et al. [1993] proposed an object-algebra operator, called `materialize (Mat)`, to enable algebraic optimization of path expressions (e.g., e.dept().plant()). The purpose of `Mat` is to represent the computation of each inter-object reference (i.e., path link) explicitly, allowing a query optimizer to express the materialization of multiple components as a group using a single `Mat` operator, or individually using a `Mat` operator per component. Another way to think of this operator is as a "scope definition," because it brings elements of a path expression into scope so that these elements can be used in later operations or in predicate evaluation. The scoping rules in TI's Open OODB optimizer algebra are very simple. An object component gets into scope either by being scanned (captured using the logical `Get` operator in the leaves of expressions trees) or by being referenced (captured in the `Mat` operator). Components remain in scope until a projection discards them. The materialize operator allows a query processor to aggregate all component materializations required for the computation of a query regardless of whether the components are needed for predicate evaluation (e.g., e.dept().plant()) or to produce the result of a query (e.g., e.job()) as shown in Figure 4c.

The purpose of the materialize operator is to indicate to the optimizer where path expressions are used and where algebraic transformations can be applied. In the example illustrated in Figure 4, the materialize operators can trade their positions in the query expression, with the condition that plant must be materialized before department. We presume that the location and name instance variables are similar to record fields that need not be explicitly materialized since they are brought into scope when their containing components (i.e., employee, department, plant, and job) are materialized.

Representing path expressions as sequences of algebraic materialize operators permits the optimizer to analyze all feasible permutations and cost-effective algorithms for computing a path. The following are sample logical transformation rules involving the materialize operator used in the Open OODB optimizer.

1. Commutativity of materialize and select.

   $\texttt{MAT}\ \alpha_1\ (\texttt{SELECT}\ \alpha_2\ (\beta_1)) \rightarrow \texttt{SELECT}\ \alpha_3\ (\texttt{MAT}\ \alpha_4\ (\beta_1))$.
   $\texttt{SELECT}\ \alpha_1\ (\texttt{MAT}\ \alpha_2\ (\beta_1)) \rightarrow \texttt{MAT}\ \alpha_3\ (\texttt{SELECT}\ \alpha_4\ (\beta_1))$; if α_1 is not dependent on α_2.

2. Commutativity of materialize and join.

   $\texttt{MAT}\ \alpha_1\ (\texttt{JOIN}\ \alpha_2\ (\beta_1\beta_2)) \rightarrow \texttt{JOIN}\ \alpha_3\ ((\texttt{MAT}\ \alpha_4\ (\beta_1))\beta_2)$; if α_1 only depends on β_1.
   $\texttt{JOIN}\ \alpha_1\ (\beta_1(\texttt{MAT}\ \alpha_2\ (\beta_2))) \rightarrow \texttt{MAT}\ \alpha_3\ (\texttt{JOIN}\ \alpha_4\ (\beta_1\beta_2))$; if α_1 does not depend on α_2.

3. Materialize to join.

   $\texttt{MAT}\ \alpha_1\ (\beta_1) \rightarrow \texttt{JOIN}\ \alpha_2\ (\beta_1(\texttt{GET}\ \alpha_3\ ()))$; if β_1 is a scannable collection.

4. Commutativity of contiguous materialize operators.

   $\texttt{MAT}\ \alpha_1\ (\texttt{MAT}\ \alpha_2\ (\beta_1)) \rightarrow \texttt{MAT}\ \alpha_2\ (\texttt{MAT}\ \alpha_1\ (\beta_1))$; if α_1 does not depend on α_2.

5. Collapse contiguous materialize operators.

   $\texttt{MAT}\ \alpha_1\ (\texttt{MAT}\ \alpha_2\ (\beta_1)) \rightarrow \texttt{MAT}\ \alpha_3\ (\beta_1)$.

In their work on the optimization of path expressions, Lanzelotte and Valduriez [1991] also present rules to transform a series of `implicit join` operators into an indexed scan using a path index when an index is available.

Path indexes

Substantial research on object query optimization has been devoted to the design of index structures to speed up the computation of path expressions [Maier and Stein 1986; Bertino and Kim 1989; Valduriez 1987; Kemper and Moerkotte 1994]. Figure 5 shows an optimal execution plan equivalent to the execution plan in Figure 4d for the case when the Department extent is indexed through the path department.plant().location().

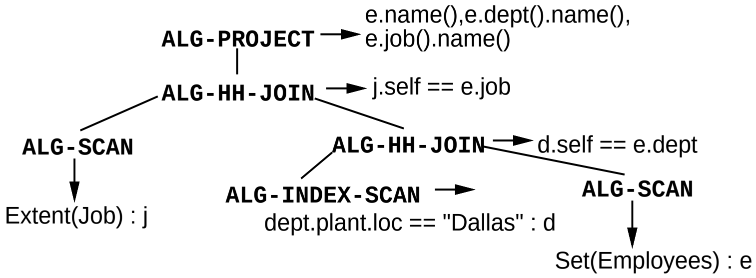

Figure 5: An optimal execution plan using a path index.

Computation of path expressions via indexes represents just one class of query-execution algorithms used in object-query optimization. In other words, efficient computation of path expressions through path indexes represents only one collection of implementation choices for algebraic operators, such as materialize and join, used to represent inter-object references. Section 4 describes a representative collection of query-execution algorithms that promise to provide a major benefit on the efficient execution of object queries. We will defer a discussion of some representative path index techniques to that section. Bertino and Kim [1989] present a more comprehensive survey of index techniques for object query optimization.

An Example

We conclude this section by completing the description of the example presented in Figure 4, optimized using the Open OODB optimizer. Recall that the materialize operators with arguments job(), dept(), and plant(), represent the logical references in the path expressions appearing in the select and project operator arguments. This query results in the query execution plan shown in Figure 4d. There are five interesting observations in comparing the algebraic query (Figure 4c) and the optimal plan (Figure 4d). First, two materialize operators are transformed to join operations and executed using hybrid-hash join. This transformation is possible because the materialize operators explicitly represent each of the path expressions' links that need to be traversed to establish the relationships between object components. Second, path expressions are computed in an order different than the one presented by the initial user query (compare where the employee-job relationship is established in Figures 4c and 4d). Reversing the order in which the links are traversed is possible because the materialize operator represents links at the logical level and the equivalence rules allow the optimizer to choose the path-computation sequence with minimal cost. Third, the plan traverses some links in a direction opposite to that of the physical pointers between the objects. This interesting and initially counterintuitive choice stems from a small extent size for the job type and a small cardinality of filtered and assembled Department-Plant objects (used in this example), which permit very efficient executions of hybrid hash join using only in-memory hash tables and no overflow files. Fourth, the placement of the assembly algorithm in Figure 4d attempts to minimize the number of plant components that have to be assembled from disk. An unfortunate choice would be to assemble first departments and then plants – typically, the number of employees is much larger than the number of departments in an organization – although it might be considered the most "natural" execution of this example query. Finally, for the department and job components, there is an explicit extent and the optimizer can place an upper bound on the number of I/O operations needed to assemble the department and job components of an employee object. This example illustrates that naive pointer chasing in object-query processing (i.e., goto's on disk) may be suboptimal in some cases. Therefore, value-based set-matching algorithms similar to those used in relational query optimization are also relevant in object-query optimization.

4 Query Execution

The relational DBMSs benefit from the close correspondence between the relational algebra operations and the access primitives of the storage system. Therefore, the generation of the execution plan for a query expression basically concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In OODBMSs the issue is more complicated due to the difference in the abstraction levels of behaviorally defined objects and their storage. Encapsulation of objects, which hides their implementation details, and the storage of methods with objects pose a challenging design problem which can simply be stated as follows: "At what point in query processing should the query optimizer access information regarding the storage of objects?" In [Straube and Özsu 1991], which follows the methodology depicted in Figure 1, access to this information is left to an *object manager*. Consequently, the query-execution plan is generated from the query expression that is obtained at the end of the query-rewrite step by mapping the query expression to a well-defined set of object-manager interface calls. The object-manager interface consists of a set of execution algorithms. This section reviews some of the execution algorithms that are likely to be part of future high-performance object-query execution engines.

A query-execution engine requires three basic classes of algorithms on collections of objects: (collection) *scan*, *indexed scan*, and *collection matching*. Collection scan is a straightforward algorithm that sequentially accesses all objects in a collection. We do not discuss this algorithm further. Indexed scan allows efficient access to selected objects in a collection through an index. It is possible to use an object's field or the values returned by some method as a key to an index. Also, it is possible to define indexes on values deeply nested in the structure of an object (i.e., path indexes). In this section we mention a representative sample of path-index proposals. Set-matching algorithms take multiple collections of objects as input and produce aggregate objects related by some criteria. Join, set intersection, and assembly are examples of algorithms in this category.

4.1 Path Indexes

Indexes are an essential component in database systems to speed up the evaluation of queries. Indexes enable fast computation of queries involving highly selective predicates and are useful access paths to accelerate the computation of set-matching operations (e.g., join).

Support for path expressions is a feature that distinguishes object-oriented from relational queries. Many indexing techniques designed to accelerate the computation of path expressions have been proposed [Maier and Stein 1986; Valduriez 1987; Bertino and Kim 1989]. *Access support relations* [Kemper and Moerkotte 1994] are a general technique to represent and compute path expressions. In their evaluation, Kemper and Moerkotte provide initial evidence that the performance of queries executed using access support relations improves by about two orders of magnitude over queries that do not use access support relations. A system using access support relations needs to also consider the cost of maintaining them in the presence of updates to the underlying base relations. For a detailed description of this topic, please refer to the chapter by Kemper and Moerkotte in this book. Maier and Stein [1986] proposed a path indexing technique for the GemStone OODBMS that creates an index on each class traversed by a path. This technique was also proposed for the Orion OODBMS [Bertino and Kim 1989]. Access support relations generalize this indexing technique to allow set-valued path expressions. In addition to indexes on path expressions it is possible to define indexes on objects across their type inheritance. Kim et al. [1989] provide a thorough discussion of such indexing techniques through inheritance.

4.2 Set Matching

In the subsequent discussion we assume that we have two sets of objects R and S that stand in a many-to-one relationship from R to S. We assume that R and S are stored as separate disk files, and that the objects in R contain an OID to their related object in S.

Hybrid-Hash Join

An effective object-query execution engine needs a generic, value-based join algorithm for three reasons. First, naive pointer traversal is not always the best algorithm to compute the join of R and S [Shekita and Carey 1990]. Second, if a query involving the join of R and S applies a highly selective predicate to the objects in S, then it is often more efficient to compute the join of R and S in the direction opposite to the direction of the pointers. Similarly, when inter-object references do not include inverse relationships, a value-based join is an effective way to compute the inter-object references between two related sets in the direction opposite to the pointers. Third, a slight modification of a generic join algorithm can be used to efficiently compute other set matching operations (e.g., intersection) [Graefe 1993].

The hybrid-hash algorithm applies the divide-and-conquer principle to the problem of computing a join. The potentially large input sets are recursively partitioned into smaller subfiles (buckets), each of which may fit entirely in memory, using a hash function on the join attribute; at the end of this stage, each subfile contains objects from the input sets that may potentially join. Each pair of subfiles is joined to produce the result. The hybrid-hash join method takes advantage of the main memory available by performing the first sub-join while building the subfiles for subsequent manipulation. The standard hybrid-hash algorithm consists of $B + 1$ steps where

$$B = \max(0, \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil),$$

where $|R|$ and $|M|$ are the sizes of relation R and main memory available (in pages), respectively, and F is a space-overhead factor for hashing (typically, 1.2). During the first step, R and S are read into memory and partitioned into $B + 1$ compatible subsets, R_i, S_i, $0 \leq i \leq B$, through a hash function on the joining attribute. In addition, the first subsets, R_0 and S_0 are joined at this time while the remaining subsets are written to disk. The remaining B steps repeat the previous stage on subset pairs R_i, S_i, $1 \leq i \leq B$ by reading them into memory and joining them.

Pointer-Based Hybrid-Hash Join

The pointer-based hybrid-hash algorithm [Shekita and Carey 1990] is used in cases when each object in R contains a pointer (PID, physical identifier) to an S object. The algorithm is similar to standard hybrid-hash join and uses the following three steps. First, R is partitioned in the same way as in the hybrid hash algorithm, except that it is partitioned by PID values rather than by join attribute. The set of objects S is not partitioned. Second, each partition R_i of R is joined with S by taking R_i and building a hash table for it in memory. The table is built by hashing each object $r \in R$ on the value of its pointer (PID) to its corresponding object in S. As a result, all R objects that reference the same page in S are grouped together in the same hash-table entry. Third, after the hash table for R_i is built, each of its entries is scanned. For each hash entry, the corresponding page in S is read and all objects in R that reference that page are joined with the corresponding objects in S.

An important difference between this algorithm and standard hybrid-hash is that R is the only set that is partitioned, and as such it always plays the role of the *inner* set. This is necessary because the direction of the pointers is from R to S. Shekita and Carey [1990] showed that when R is significantly larger than S, standard hybrid-hash may outperform pointer-based hybrid-hash, therefore, an OODBMS can benefit by supporting both algorithms.

Assembly

The `assembly` operator [Keller et al. 1991] is a generalization of the pointer-based hash-join algorithm for the case when we need to compute a multi-way join. There is a difference between assembly and n-way pointer joins in that assembly does not need the entire collection of root objects to be scanned before producing a single result. Assembling a complex object rooted at objects of type R containing object components of types S, U, and T, is analogous to computing a four-way join of these sets.

Instead of assembling a single complex object at a time, the assembly operator assembles a *window*, of size W, of complex objects simultaneously. As soon as any of these complex objects becomes assembled and passed up the query-execution tree, the assembly operator retrieves another one to work on. Using a window of complex objects increases the pool size of unresolved references and results in more options for optimization of disk accesses. Due to the randomness with which references are resolved, the assembly operator delivers assembled objects in random order up the query execution tree. This behavior is correct in set-oriented query processing, but may not be for other collection types such as lists.

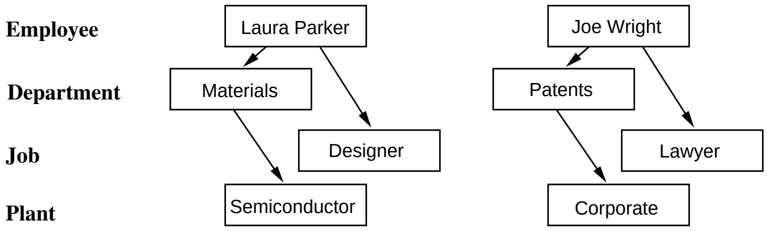

Figure 6: Two assembled complex objects.

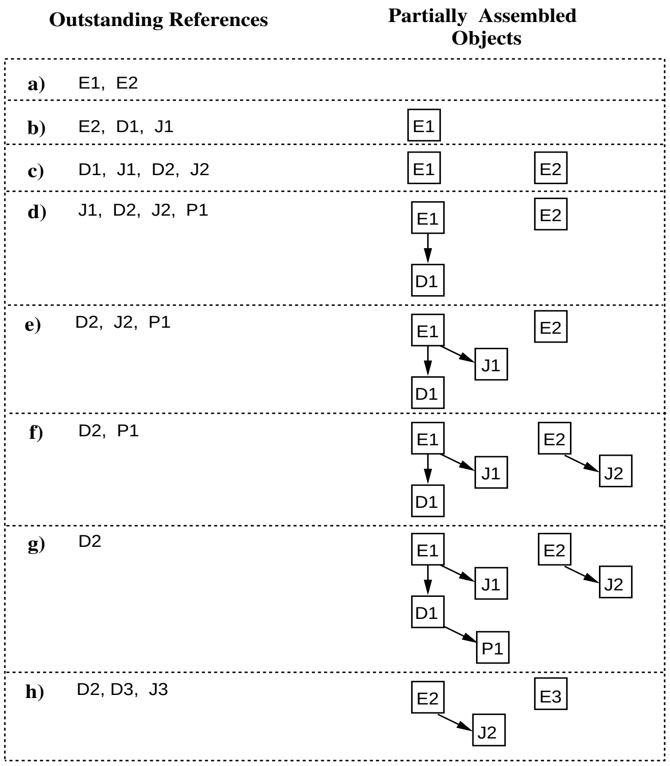

Figure 7: An assembly example.

To illustrate the behavior of assembly, consider the following example[5] which assembles a set of employee objects with the structure shown in Figure 6. Suppose that assembly is using a window of size 2. The assembly operator begins by filling the window with two (since $W = 2$) employee references from the set (Figure 7a). The assembly operator begins by choosing among the current outstanding references, say $E1$. After resolving (fetching) $E1$, two new unresolved references are added to the list (Figure 7b). Resolving $E2$ results in two more references added to the list (Figure 7c), and so on until the first complex object is assembled (Figure 7g). At this point, the assembled object is passed up the query-execution tree, freeing some window space. A new employee reference, $E3$, is added to the list and then resolved, bringing two new references $D3$, $J3$ (Figure 7h).

The objective of the assembly algorithm is to simultaneously assemble a window of complex objects. At each point in the algorithm, the outstanding reference that optimizes disk accesses is chosen. There are different orders or schedules in which references may be resolved. Keller et al. [1991] study the performance of three reference resolution

[5]This example is taken from [Keller et al. 1991] with slight modifications in notation.

algorithms for assembly: depth-first, breath-first, and elevator. The results indicate that elevator outperforms depth-first and breath-first under several data-clustering situations.

5 Conclusions

This chapter reviewed some of the main contributions to object-database query processing to date. In the area of query processing architectures, we expect future object-query architectures to continue to bear a strong similarity with relational and extended-relational query-processing architectures. The refinement, design, and better understanding of internal interfaces among the modules that constitute a query-processing system will continue to be important research challenges. Building query processors under highly modular and extensible architectural frameworks will accelerate the technology transfer process of this rapidly evolving technology into working systems. The EREQ [Maier 1992], TI Open OODB [Wells et al. 1992], Volcano [Graefe and McKenna 1993], and TIGUKAT [Peters et al. 1992] are projects in this direction.

In the area of object-query optimization, some fairly comprehensive cost-based query optimizers have started to appear. Research into query-optimizer generator toolkits (e.g., [Graefe and McKenna 1993]), and rule-based query optimization has moved beyond the proof-of-concept stage. A few non-relational query optimizers for object-oriented [Blakeley et al. 1993] and scientific databases [Wolniewicz and Graefe 1993] have been built with the help of such generators. Given that query optimizers are fairly complex software systems to build, we expect to see an increase in the use of optimizer generators to help build new query optimizers in the years to come. We also expect an increase in the scope of the optimization tasks for which optimizer generators are used. Cost-based query optimization has proven to be still an effective approach for the optimization of object queries. We believe that complexity and size of the search space are not immediate road blocks in the effectiveness of cost-based query optimization for complex object queries. Rather, the deficiency of cost- and selectivity-estimation techniques and error propagation will be immediate concerns when scaling the cost-based query optimization approach to complex queries. The judicious incorporation of heuristic pruning, randomization techniques, and the combination of partial query execution (to obtain exact cost and selectivity estimates) with cost-based optimization are important areas of future research.

In the area of query execution, perhaps a main contribution by object-oriented DBMSs has been the development of indexing techniques over path expressions and inheritance type hierarchies to speed up access to collections of complex objects. It has also become clear that join and set-matching execution algorithms initially developed for relational databases will continue to play an important role in the development of high-performance object-query execution engines. Naive pointer traversal (goto's on disk) is not always optimal when querying large collections of objects. New execution algorithms to efficiently access complex objects, such as object assembly and pointer-based joins, have been developed.

Future DBMSs will be hybrid open systems that combine the best capabilities of existing DBMSs. These future DBMSs will be as effective as existing relational DBMSs in the processing of queries, and provide rich data modeling and high-performance object-at-a-time access as existing OODBMS. In other words, relational DBMSs will try to provide better support for objects and OODBMS will try to provide better support for queries. Database-query processing will continue to be an exciting and challenging research area.

Acknowledgements

We would like to thank Goetz Graefe and Gail Mitchell for their careful reading and useful comments on the first version of this chapter. M. Tamer Özsu's research is supported by Natural Sciences and Engineering Research Council (NSERC) of Canada under the research grant OGP0951. José Blakeley's research is sponsored by the Advanced Research Projects Agency under ARPA Order No. A016 and managed by the U.S. Army Research Laboratory under contract DAAB07-90-C-B920.

References

ABITEBOUL, S., AND BEERI, C. 1987. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA.

ALHAJJ, R., AND ARKUN, M.E. 1993. A query model for object-oriented databases. In *Proc. 9th Int. Conf. on Data Engineering*, 163–172.

ATKINSON, M., BANCILHON, F., DEWITT, D., DITTRICH, K., MAIER, D., AND ZDONIK, S. 1989. The object-oriented database system manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 40–57.

BEERI, C., AND KORNATZKY, Y. 1990. Algebraic optimization of object-oriented query languages. In *Proc. 3rd Int. Conf. on Database Theory*. Springer Verlag, 72–88.

BERTINO, E., AND KIM, W. 1989. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering 1*, 2, (June), 196–214.

BLAKELEY, J. 1991. DARPA open object-oriented database preliminary module specification: Object query module. Technical Report, Texas Instruments, (Dec.).

BLAKELEY, J. 1994. OQL[C++]: Extending C++ with an object query capability. In this book.

BLAKELEY, J., MCKENNA, W., AND GRAEFE, G. 1993. Experiences building the Open OODB query optimizer. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 287–296.

CAREY, M., DEWITT, D., AND VANDENBERG, S. 1988. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 413–423.

CLUET, S., AND DELOBEL, C. 1992. A general framework for the optimization of object-oriented queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 383–392.

DEUX, O. ET AL. 1991. The O_2 system. *Comm. of the ACM 34*, 10, (Oct.), 34–48.

DOGAC, A., OZKAN, C, ARPINAR, B., OKAY, T., AND EVRENDILEK, C. 1994. METU object-oriented DBMS. In *Advances in Object-Oriented Database Systems*, A. Dogac, M.T. Özsu, A. Biliris, T. Sellis, Eds. Springer-Verlag.

FREYTAG, J. 1987. A rule-based view of query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 173–180.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Computer Surveys 25*, 2, (June), 73–170.

GRAEFE, G., AND DEWITT, D. 1987. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 160–172.

GRAEFE, G., AND MAIER, D. 1988. Query optimization in object-oriented database systems: The REVELATION project. Technical Report CS/E 88-025, Oregon Graduate Center.

GRAEFE, G., AND MCKENNA, W. 1993. The Volcano optimizer generator. In *Proc. 9th Int. Conf. on Data Engineering*, 209–218.

GRAEFE, G., AND WARD, K. 1989. Dynamic query evaluation plans. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 358–366.

HAAS, L., CODY, W., FREYTAG, J., LAPIS, G., LINDSAY, B., LOHMAN, G., ONO, K., AND PIRAHESH, H. 1989. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 377–388.

IOANNIDIS, Y., AND CHA KANG, Y. 1990. Randomized algorithms for optimizing large join queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 312–321.

IOANNIDIS, Y., NG, R., SHIM, K., AND SELLIS, T. 1992. Parametric query optimization. In *Proc. 18th Int. Conf. on Very Large Data Bases*, 103–114.

IOANNIDIS, Y., AND WONG, E. 1987. Query optimization by simulated annealing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 9–22.

JENQ, B., WOELK, D., KIM, W., AND LEE, W.-L. 1990. Query processing in distributed ORION. In *Advances in Database Technology — EDBT'90*. Springer-Verlag, 169–187.

KELLER, T., GRAEFE, G., AND MAIER, D. 1991. Efficient assembly of complex objects. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 148–157.

KEMPER, A., AND MOERKOTTE, G. 1994. Physical object management. In this book.

KIFER, M., KIM, W., AND SAGIV, Y. 1992. Querying object oriented databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 393–402.

KIFER, M., AND WU, J. 1989. A logic for object-oriented programming (Maier's O-logic: Revisited). In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 379–393.

KIM, W. 1989. A model of queries for object-oriented databases. In *Proc. 15th Int. Conf. on Very Large Data Bases*, 423–432.

KIM, W., KIM, K., AND DALE, A. 1989. Indexing techniques for object-oriented databases. In *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovshy, Eds. ACM/Addison-Wesley, 371–292.

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The ObjectStore database system. *Comm. of the ACM 34*, 10 (Oct.), 50–63.

LANZELOTTE, R., AND VALDURIEZ, P. 1991. Extending the search strategy in a query optimizer. In *Proc. 17th Int. Conf. on Very Large Databases*, 363–373.

LEE, M., FREYTAG, J., AND LOHMAN, G. 1988. Implementing an interpreter for functional rules in a query optimizer. In *Proc. 14th Int. Conf. on Very Large Databases*, 218–229.

MAIER, D. 1986. A logic of objects. In *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, 6–26.

MAIER, D. 1992. Specifying a database system to itself. In *Specifications of Database Systems*, D. Harper and M. Norrie, Eds. Springer-Verlag.

MAIER, D., AND STEIN, J. 1986. Indexing in an object-oriented DBMS. In *Proc. 1st Int. Workshop on Object-Oriented Database Systems*, 171–182.

MITCHELL, G., DAYAL, U., AND ZDONIK, S. 1993. Control of an extensible query optimizer: A planning-based approach. In *Proc. 19th Int. Conf. on Very Large Databases*, 517–528.

MUÑOZ, A. 1994. An extensible query optimizer for the TIGUKAT objectbase management system. Master's thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, 1994.

ONO, K., AND LOHMAN, G. 1990. Measuring the complexity of join enumeration in query optimization. In *Proc. 16th Int. Conf. on Very Large Databases*, 314–325.

ORENSTEIN, J., HARADVALA, S., MARGULIES, B., AND SAKAHARA, D. 1992. Query processing in the ObjectStore database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 403–412.

ÖZSU, M., AND STRAUBE, D. 1991. Issues in query model design in object-oriented database systems. *Computer Standards & Interfaces 13*, 157–167.

ÖZSU, M., STRAUBE, D., AND PETERS, R. 1993. Query processing issues in object-oriented knowledge base systems. In *Emerging Landscape of Intelligence in Database and Information Systems*, F. Petry and L. Delcambre, Eds., JAI Press.

PETERS, R., LIPKA, A., ÖZSU, M., AND SZAFRON, D. 1993. An extensile query model and its languages for a uniform behavioral object management system. In *Proc. 2nd International Conference on Information and Knowledge Management*, 403–412.

PETERS, R., ÖZSU, M., AND SZAFRON, D. 1992. TIGUKAT: An object model for query and view support in object database systems. Tech. Rep. TR92-14, Department of Computing Science, University of Alberta, (Oct.).

SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 23–34.

SHAPIRO, L. 1986. Join processing in database systems with large main memories. *ACM Transactions on Database Systems 11*, 3 (Sept.), 239–264.

SHAW, G., AND ZDONIK, S. 1990. A query algebra for object-oriented databases. In *Proc. 6th Int. Conf. on Data Engineering*, 154–162.

SHEKITA, E., AND CAREY, M. 1990. A performance evaluation of pointer-based joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 300–311.

STONEBRAKER, M., ROWE, L., LINDSAY, B., GRAY, J., CAREY, M., BRODIE, M., BERNSTEIN, P., AND BEECH, D. 1990. Third-generation data base system manifesto. *ACM SIGMOD Record 19*, 3, (Sept.), 31–44.

STRAUBE, D., AND ÖZSU, M. 1990a. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems 8*, 4, (Oct.), 387–430.

STRAUBE, D., AND ÖZSU, M. 1990b. Type consistency of queries in an object-oriented database system. In *Proc. ECOOP/OOPSLA '90 Conference*, 224–233.

STRAUBE, D., AND ÖZSU, M. 1991. Execution plan generation for an object-oriented data model. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, C. Delobel, M. Kifer, and Y. Masunaga, Eds. Springer-Verlag, 43–67. (A full version will appear in *IEEE Transactions on Knowledge and Data Engineering*).

SWAMI, A. 1989. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 367–376.

ULLMAN, J. 1988. *Principles of Database and Knowledge Base Systems*. Computer Science Press.

VALDURIEZ, P. 1987. Join indices. *ACM Transactions on Database Systems 12*, 2 (Dec.), 218–246.

VANDENBERG, S., AND DEWITT, D. 1991. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, (June), 158–167.

WELLS, D., BLAKELEY, J., AND THOMPSON, C. 1992. Architecture of an open object-oriented database management system. *Computer 25*, 10 (Oct.), 74–82.

WOLNIEWICZ, R., AND GRAEFE, G. 1993. Algebraic optimization of computations over scientific databases. In *Proc. 19th Int. Conf. on Very Large Databases*, 13–24.

YU, L., AND OSBORN, S. 1991. An evaluation framework for algebraic object-oriented query models. In *Proc. 7th Int. Conf. on Data Engineering*, 670–677.

ZANIOLO, C. 1983. The database language GEM. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 207–218.