Extensible Query Processing in an Object-Oriented Database

Gail Anne Mitchell

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-16
May 1993

Extensible Query Processing in an Object-Oriented Database [†]

by
Gail Anne Mitchell
B. A., Cedar Crest College, 1972
M. S., The Pennsylvania State University, 1984
Sc. M., Brown University, 1990

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May, 1993

[†]Partial support for this research was provided by the Office of Naval Research (contracts N0014-88-K-0406 and N00014-91-J-4052), by the Defense Advanced Research Projects Agency (contract DAAB-07-91-C-Q518), and by Texas Instruments.

This dissertation by Gail Anne Mitchell
is accepted in its present form by the Department of
Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Paris C. Kanellakis

Recommended to the Graduate Council

Date _____ _____

Umeshwar Dayal

Date _____ _____

Peter Wegner

Approved by the Graduate Council

Date _____ _____

Abstract

In this thesis we address the problem of providing efficient processing of queries in the extensible environment induced by object-oriented databases. We define a framework for query processing in an object-oriented database and develop designs for major components of this framework. The framework encompasses an object-oriented data model, an algebra to query over that model, transformation rules for the algebra, an internal representation for queries expressed in the algebra, a cost model for analyzing query expressions, and an architecture for an extensible query optimizer. The major contributions of this thesis are an algebra and transformation rules, a representation, and an architecture for extensible query optimization. We show how these components fit into the framework and interact with each other.

The EQUAL query algebra presented in this thesis is the first query algebra for object-oriented database systems to be completely consistent with data abstraction, and one of the few to propose operations for the creation and manipulation of objects with identity. We present transformation rules for this algebra, and a theory of equivalence for query expressions involving object-building operations. We also define an internal representation for query expressions that is annotated with information that can be used during query optimization, and can be extended to represent new operations and annotations.

We present Epoq, a novel approach to extensible query optimization. An Epoq optimizer integrates diverse strategies for controlling the optimization of a query and adapts to the addition of new strategies for control. We give a formal basis for this approach and specify an architecture that embodies the approach. We define the control problem for extensible query optimization and present a solution to this problem that is based on planning the optimization process. The planning-based control can combine the optimization strategies in different ways depending on the characteristics of the query being processed. The architecture and control are illustrated with an extensive example optimizer that integates optimization strategies described in the literature with strategies developed specifically for EQUAL.

Acknowledgements

This dissertation would have been impossible without the encouragement and support of the members of my committee. I couldn't have asked for a better research collaborator than Umesh Dayal — our common interests and his insightful direction led to research that was far more than I ever anticipated and to a friendship that I value greatly. Paris Kanellakis was the perfect thesis advisor — he gave me exactly the right amount of guidance and encouragement as I labored over the writing of this dissertation. I also appreciate the constant support of Peter Wegner — his critical comments greatly improved the presentation of this document.

I'm thankful for the many friends, old and new, who helped guide me through the past six years. I can't possibly thank everyone individually but do want to thank Jim Shaw for encouraging me to go back to school; Richard Hughey for listening and helping me through the first two years; and Scott Meyers for getting me working in databases again and for putting up with me for six years.

My research was helped greatly by the people in the database group at Brown. In particular, the long (sometimes heated) discussions with Page Elmore Evett helped me gain a better understanding of the context of my work and resulted in a lasting friendship. Thanks also to Ted Leung, Peter Tan, George Lo, Solveig Bjornestad, and Wayne Wong for their valuable comments as they worked (or are still working) on different of the implementation projects for this thesis.

Finally, I'd like to thank my family. Thanks to my parents, Pat and Fred Mitchell, for always believing in me and encouraging me to pursue lofty goals. Thanks to my grandparents, Dan and Marguerite Turner, for many great dinners and discussions and for providing a place for uninterrupted study. Thanks to my son, Jim, for recognizing when to stay out of my way and when to give hugs. And thanks to my husband, Stan, for everything.

Credits

The research reported in this dissertation is the result of various collaborations with Stanley B. Zdonik and Umeshwar Dayal. The EQUAL query algebra and related results in query equality and equivalence were developed under the supervision of Zdonik. My optimization results were developed in collaboration with Zdonik and Dayal.

The presentation in this dissertation is based in part on the published papers [105, 108, 132, 133, 134][1], an invited report [154], and Brown University reports [104, 107]. The publications [132, 133, 134, 154] report on my work with Zdonik; the others on my work with Dayal and Zdonik. Mitchell (Shaw) is the primary author on all papers with the exception of [154] which describes primarily the work of Zdonik. The papers map to the dissertation as described in the following annotated bibliography:

[133] This paper is an early description of the EQUAL[2] query algebra presented at the 2nd International Workshop on Database Programming Languages. The research presented here maps to Chapter 3.

[134] This paper is a full description of the EQUAL algebra presented at the 1990 Data Engineering Conference. The presentation of Sections 3.2.1 and 3.2.2 is taken from this paper.

[132] This paper is a description of the work in query equivalence presented at the 1st International Conference on Deductive and Object-Oriented Databases. The presentation of Sections 3.3 and 4.4.3 is taken from this paper.

[154] The ENCORE data model and EQUAL query algebra are summarized in this IEEE Data Engineering Bulletin report. This maps to Chapter 3 in general, and in particular to Section 3.1.

[107] This technical report on the problems faced in object-oriented query optimization is the basis for Chapter 4 of this dissertation. This material, along with parts of Chapter 2, will also be published as [106].

[104] This working paper on query representation is the basis for the presentation in Chapter 7.

[108] This paper, presented at the 1992 Hawaii International Conference on System Sciences, is a preliminary description of our work on query optimization. This work maps to the presentation of Sections 5.1 and 5.2.

[105] This paper describing the control problem and planning-based approach to control has been accepted for publication in the Proceedings of the 19th International Conference on Very

[1] Most of the research related to the EQUAL query algebra is published under my former name, Gail Mitchell Shaw.

[2] The name EQUAL is not used in publication, though, until [154].

Large Data Bases (August 1993). The paper is mostly extracted from Section 5.1.3 and Chapter 6. The paper also outlines the example optimizer of Chapter 8.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Object-oriented database systems provide a highly extensible environment that challenges traditional ideas about query processing. Query languages for object-oriented databases must incorporate support for object identity, complex structures, abstract data types, and subtyping, for example, as features supported by the object-oriented model. Queries involve a changing variety of database types and access to objects through arbitrary methods defined by abstract data types.

Efficient processing of queries demands an optimizer that can adapt to the ever-changing environment of the object-oriented database system. New abstractions may be described and related to existing abstractions by new axioms that must be considered by a query rewrite system. New types may be supported by new methods for accessing data that must be considered when generating an execution plan for the query. New types may also be accompanied by new techniques for optimizing queries involving those types. The optimizer must have some way to integrate these new techniques with existing optimization processing.

In this thesis, we address the problem of providing efficient processing of queries in the extensible environment induced by object-oriented databases. The major contributions of this thesis are:

- an object-oriented query algebra with operations that access instances of abstract data types, and operations to create and manipulate objects with identity
- an annotated representation for queries that treats query operations, predicates, and arbitrary methods over objects uniformly, thus supporting cost– and heuristic–based transformations, and transformations involving nested subqueries
- a novel approach to extensibility in query optimization that integrates diverse strategies for controlling the optimization of queries and adapts to incorporate new control strategies
- an architecture for query optimization that embodies our approach to optimizer extensibility

These contributions, and ancillary contributions, are discussed in more detail in Section 1.3.

The work described in this thesis was initiated in order to define querying and query optimization in the ENCORE object-oriented database system [157, 154]. Although we describe the work in the context of this model, our results in most cases apply more globally. The query algebra has already been used as a target for optimization of a declarative language [31], and our architecture for query optimization presents a new approach to extensibility that applies to any database model.

An object-oriented database model, such as ENCORE, is extensible in that it provides, through abstract data types, the ability to add new types with new interface methods and implementations. In other words, the data abstraction capability can be used to describe new functionalities for the database system. Other components of a query processing system must support the extensibility

of the model. The internal representation of the query must be able to correctly model new data types and operations. A cost model may require new computations to handle the new types and operations. New axioms may describe the semantics of the new operations, and must be represented by transformations in an optimizer. The implementations of types are new operations for database access that must also be considered by a query optimizer. In general, a query optimizer must adapt to use new information when it processes queries involving the new types.

The ENCORE query algebra (EQUAL) presented here supports the object-oriented database model by querying database objects only through their interface [134]. The query representation presented in Chapter 7 supports additions to the language and model. The representation is annotated with information about the query (e.g., cost and typing information). The collection of annotations can be extended to support new kinds of information about queries that may be required by an optimizer.

The Epoq[1] architecture for query optimization presents a new approach to optimizer extensibility. Traditionally, extensibility in query optimization refers to the ability to define new data types and methods, and to add new rewrite rules describing transformations of query expressions involving the new types and methods. Traditional extensible optimizers, though, define a fixed strategy for controlling the search for rewrite rules to apply to a query. The Epoq architecture supports the traditional kind of extensibility and additionally provides the ability to add new strategies for controlling the query transformation process.

An Epoq optimizer is a collection of modules, each of which represents a particular strategy for manipulating query expressions. The Epoq control integrates these different strategies into a single optimizer. The collection of strategies can be extended by adding new modules, and the control over the execution of these modules can be extended in response.

It is interesting to notice that our view of extensible optimization is strongly related to integration. The optimizer control integrates different query processing strategies to form a single optimizer. These strategies are encapsulated in modules, the internals of which are hidden to the optimizer. The only requirement is that the modules obey an interface defined by the optimizer control. Thus foreign optimizer components can be integrated into an Epoq optimizer along with components that are explicitly designed for Epoq.

1.1 Preliminaries

1.1.1 Query Processing

Query processing refers to the expression and efficient execution of a request for information from a database. A query is expressed in a high level language, and a query optimizer is responsible for translating that expression into an efficient plan for accessing the database. A query optimizer is not generally expected to discover the most efficient way to process a query.

The major problem in query processing is that a query refers to large amounts of data stored on relatively slow media. The query is expressed in a high level language that leaves operator implementations and even choice of and ordering of query operations open. The optimizer's job is to find a way to retrieve the requested data. An optimizer must bind operators to implementations and find orderings for operations that will most quickly answer the query. Often an optimizer uses heuristics to formulate initial plans for data retrieval. Eventually, however, the physical characteristics of the data and access methods must be considered. For example, indexes, clustering and specialized access methods can affect the speed of responding to a query and must be discovered

[1] Pronounced as in epoch, Epoq provides extensible processing and optimization of queries.

and used effectively by the optimizer. These aspects of the query can be analyzed with a cost model and used to eliminate potentially poor plans for data retrieval.

The ability to do automatic query optimization is one of the strengths of relational database systems. Relational query optimizers exploit the simple semantics of the model and the fixed sets of operators, storage structures, and implementation techniques for the operators. The designs of such optimizers differ and are specific to the system in which the optimizer is built [37, 149]. These optimizers generally embody a set of pre-defined manipulations on some internal query representation. These manipulations apply built-in heuristics to guide the discovery of efficient strategies for database access. The access strategies are evaluated according to some cost formula, and a best strategy is selected. The heuristics that are applied, the algorithms for searching for strategies, and the cost model upon which strategy evaluation is based, are all fixed and specific to the particular optimizer.

The approach to query processing taken in this thesis is to provide a high-level algebraic language for query expression and an architecture for designing optimizers that can manipulate queries expressed in the language, utilize a cost model, and incorporate techniques for data access. The language can be directly manipulated using algebraic transformation rules, and we present some rules for doing this. We do not present specific access techniques, such as indexes or clustering strategies, or a detailed cost model, but instead provide a framework that can incorporate query manipulation, access techniques and a cost model.

1.1.2 Extensibility

Extensibility in database systems addresses the need to respond to the varied requirements of new and different database applications. A database system is extensible when it can incorporate new functionalities as required to support an application.

Some systems extend the relational model with the ability to define new data types and operations, and new access methods for those types [64, 142]. Here, extensibility refers to the ability to add new types defining data stored in relational tables. The underlying system model (i.e. relational) remains fixed, but the kinds of information that can be stored in the model can be extended. The earliest extensible optimizers were designed to address the additional features provided by these systems [117].

Other systems give an implementor the tools to describe a database model and generate a new database system from that description [14, 24, 61]. The goal of this generative, or toolkit, approach to database extensibility is to allow a specialized database system to be built to meet the needs of a particular application. Extensibility in such systems refers to the ability to define data and access requirements for the new database model. An existing system can be extended by adding information to the database description, and generating another system that includes the new information.

Optimizers for extensible database systems are generally based on transformation rules for a set of operators defined in the system [45, 54]. Extensibility of these optimizers results from the ability to augment the set of transformation rules. These transformations are applied to a query expression to generate equivalent, and hopefully more efficient, forms of the expression. A transformed expression often corresponds to many executable plans for database access. These plans are compared according to a cost model (which might also be extensible). The quality of the result produced by an optimizer depends on the completeness of the defined set of transformation rules, as well as the cost model. The efficiency of the optimizer itself depends on the control strategies for selecting which transformations to apply, and when to apply them.

A more open approach to database extensibility allows an existing system to be modified by

"plugging-in" new features. Using this approach, a general database system can be customized to support specialized applications. The new features could be added through re-compilation, through linking, or even while the system is running. In a Starburst optimizer, for example, new rewrite rules can be enabled or disabled while the system is executing [115].

This thesis takes an open approach to extensibility. The object-oriented model allows the description of new types and operations for database access. The algebraic operations access objects only through the interface defined for their types and thus are general enough to incorporate the new types automatically. Also, if desired, the object-oriented model also provides the ability to define new operations for the algebra. Our query representation is general enough to model any operations defined in the model, and new annotations can be added to the representation to capture information that may be required for the new types and operations. The Epoq architecture describes a modular system for query optimization in which modules can be added or deleted with well-defined, localized effects on other modules in the system. An optimizer built using this architecture is thus extensible, and could be implemented so that changes can be made at any time. Many of the features we describe for an Epoq optimizer could be generated, although we do not address the mechanics of such generation here.

1.1.3 Object-Oriented Database Systems

Object-oriented databases respond to the needs of a variety of database applications by providing flexible database modelling capabilities [9, 12, 98, 102, 157]. Object-oriented refers to the fact that the data modelling features for such a database are drawn from the area of object-oriented programming languages.

There is much discussion about what constitutes a model for an object-oriented [8] or next generation [141] database system. Although there is no accepted common object-oriented database model, most agree that the data model should provide features such as data abstraction or encapsulation, complex object descriptions, types and subtyping, and object identity. However, different systems often place different meanings on these terms and provide differing amounts of support for some of the features. For example, often object-oriented database systems allow the definition of complex objects that are not instances of abstract types [13]. The result is that the implementation of the object is exposed for manipulation by a query.

The ENCORE model used in this thesis defines all types as abstract, thus every object has an interface that is separate from its implementation and all new types are indistinguishable from the built-in types. The model also supports subtyping relationships between the types. In the model, everything is an object. We use this model because it cleanly separates the data model from its implementation. As a result, it can be used to simulate most other object-oriented database models.

Object-oriented database systems are also extensible systems; the extensibility of object-oriented databases is founded on the ability to extend the data model through abstract data types and the inheritance/subtyping structure. In particular, the ability to define new abstract data types provides extensibility in the database modelled by the abstract data types and places a requirement on the database system to support the new model. Extensibility in object-oriented database systems is thus more fundamental to the underlying database model than in extensible relational systems and is also more dynamic than the extensibility provided by toolkit systems.

Query optimization for object-oriented systems has followed two basic directions. One approach focuses on specific techniques that solve particular optimization problems. For example, one highly visible problem in optimization of object-oriented expressions is path expressions. Such expressions imply a navigation through objects to find the end of a path. Research in this area includes defining indexes for paths [19, 79, 97, 111], optimization in the presence of arbitrary methods along the path

[20, 60, 62, 78], and the use of clustering and other storage information to determine path accesses [31, 75, 91].

An alternative approach is to define a complete system for optimization. The application of algebraic transformations has formed the basis for the design of many query optimizers for object-oriented databases [49, 112, 115, 145]. Algebraic transformations, however, do not always provide ways to deal with such problems as path expressions and method invocations. We have also found that manipulations involving objects with identity complicate the definition of the equivalence of two query expressions [132] and complicate the application of transformation rules. Other proposals for optimizers allow combinations of algebraic transformation with other strategies for optimization [81, 89]. The ability to support a number of different approaches to query transformation best provides the extensibility needed for object-oriented systems and is the basis for the Epoq architecture presented in this thesis.

1.2 Scope of This Thesis

In this thesis we describe a framework for query processing in an object-oriented database. We have identified the following components as integral to the efficient processing of queries:

- a data model
- a query language
- transformation rules for queries expressed in that language
- a canonical representation for queries
- a cost model
- an optimizer

Query processing, in general, involves interactions between these components. The data model and query language directly determine the queries that can be expressed. The transformation rules, query representation and cost model support an optimizer in finding efficient ways to process those queries.

This thesis focuses on an algebra and transformation rules, an internal query representation, and an architecture for extensible optimization as our major contributions in the area of query processing in object-oriented systems. We show how these components interact with each other and with a data model and cost model.

We base our processing system on the ENCORE database model [157]. The model is based on persistent objects (with identity) as instances of abstract data types, and provides other key features of object-oriented systems such as subtyping, late-binding, and complex structures. We believe that the modelling capabilities provided by abstract data types allow the simulation of modelling features of most other proposed object-oriented models, and that ENCORE therefore provides a strong basis for the discussion of query processing in object-oriented models in general.

In this thesis we build on the ENCORE model by adding a query algebra, transformation rules, an internal query representation, and an architecture for extensible query optimization.

The EQUAL query algebra is designed to support the modelling features of ENCORE. EQUAL generalizes the operations of the relational algebra to allow queries involving method invocations on objects, and to provide operations to manipulate objects with complex structure and identity. The EQUAL operators are defined as methods of the ENCORE parameterized type **Set**, and are

used to construct queries over sets of ENCORE objects. All queries are strongly typed, and can be statically type-checked. The result of any query is a new database object that is an instance of a parameterized type. This means that we can build alternative paths to the same object, and also that we might build objects that are considered to be duplicates. However, our algebra does not build recursive objects since we do not want to consider the optimization of such queries at this time.

We specifically chose to express queries in an algebra because of the potential for an optimizer to use rewrite rules to manipulate the ordering of operations. However, in order to define rewrite rules for operations that create new objects, it is first necessary to define what is meant by equivalence of query expressions. We define three different meanings for equivalence in the presence of object-creating operations, and for each kind of equivalence we give some transformation rules for EQUAL expressions. We also give, in Appendix A, an inventory of rules that satisfy the most restrictive form of equivalence.

The major portion of this thesis is devoted to query optimization. The original intention of our research was to provide specific strategies for addressing common optimization problems in the object-oriented model. However, as we examined other work in this area we came to believe that the optimization problems are not going to be solved by a single, all-encompassing strategy. Thus, we chose instead to develop an architecture for query optimization that can integrate different strategies and be extended with new strategies for optimization as they are developed. This architecture was developed under the assumption that we are processing queries in a centralized database system. We believe the optimizer architecture could be applied to distributed query optimization, but have not yet explored that area.

Strategies for query optimization can be based on heuristics or on a cost model. They can describe manipulations on a high-level language (such as EQUAL) or on access methods and query execution plans. The Epoq architecture presented in this thesis integrates all of these approaches by encapsulating specific approaches as optimizer modules. We define a standard interface for modules and a control that can determine a sequence of executions of different modules. The interface describes the I/O characteristics of a strategy, and hides the definition and execution of any particular optimization strategy. The control, in accessing a strategy through the interface, treats each strategy as a single query transformation. We thus reduce the control problem to that of determining a sequence of query transformations. We propose a solution to this problem that is based on planning [40].

The architecture is supported by an internal representation for queries which we define as an annotated operator graph. Nodes in the graph represent objects, query operations, and arbitrary methods over objects, and arcs represent relationships between the data and functions in the query (i.e., operations and methods). This particular representation is suggested for optimizers built using the Epoq architecture because of its ability to represent general algebraic operations and because of its extensibility. However, an optimizer could be built using an alternative representation, and particular strategy modules can encapsulate their own representations if desired.

We assume the existence of a cost model with cost functions that can operate over the annotated operator graph, but the development of a particular cost model is not part of this work. We show where the cost model fits within the architecture so that results of future research in cost models can be incorporated into an Epoq optimizer.

Epoq is not itself an optimizer, but is an architecture describing the required components of an object-oriented query optimizer and specifying how they interact. Although we give an example of an optimizer built within this architecture, the example is intended to serve only as an illustration of the capabilities of the architecture and not as a finished proposal for a complete object-oriented query optimizer. The development of other such optimizers would benefit from a good design

methodology. Such a design methodology is beyond the scope of this thesis.

1.3 Contributions

The major contributions of this thesis are the EQUAL object-oriented query algebra, an extensible internal representation for queries, a novel approach to extensibility in query optimization, and an architecture that embodies this approach. We discuss each of these contributions, and ancillary contributions, in the following subsections.

1.3.1 The EQUAL algebra

The Encore query algebra [133, 134] contributes to our understanding of how support for object-oriented modelling concepts affects the operations required for querying within the model. In particular, EQUAL addresses problems associated with accessing objects described by abstract data types, and with accessing objects with identity and creating new objects. The algebra can simulate complex objects by describing complex structures using abstract data types. In this way it cleanly combines access and creation of complex structures with manipulation of objects having abstract data types. The algebra is also one of the few languages that builds new objects as the result of a query.

We have developed a number of equational axioms describing how different algebraic operations are related (see Appendix A). These rules describe transformations of EQUAL query expressions to give equivalent expressions. In most systems, equivalence of query expressions under rewrite can be based on equality of data values. However, the presence of object-creating operations in EQUAL means that equivalence of query expressions must consider object identity. A contribution of this thesis is a theory of equivalence for query expressions involving object-building operations [132].

1.3.2 Annotated Query Representation

The annotated query representation generalizes previous representations by providing for the uniform representation of query operations, database objects, query predicates, and methods over objects. This allows us to completely represent the structure of queries nested as parameters to other queries. As a result, we can perform transformations on query predicates that contain subqueries. To our knowledge, no other proposed representation for queries provides for the explicit representation of subqueries within predicates.

The representation consists of data nodes, function nodes, and arcs relating these. Each component type can be annotated with information about the query that can be used by transformation rules or other processing strategies. For example, annotations might be defined on data nodes to store information about query costs, and annotations might be defined on function nodes to indicate how the function affects cost. The information can be used to compute and compare costs, and can be updated if necessary, when transforming a query.

The collection of annotations for any component of the representation is defined by a system implementor, and can be extended to incorporate new annotations. The extensibility of the collections of annotations naturally supports other extensions to the query processing system, and is not provided by other proposed representations.

1.3.3 Approach to Extensibility

Our approach to extensibility in query optimization is a primary contribution of this work. We expand the meaning of extensibility to include extending the collection of optimization strategies that can be applied to transform a query expression. Current extensible optimizers concentrate on extending the sets of operators, transformation rules, and database access methods. They provide a single set of transformation rules (to which new rules can be added) and a fixed control strategy for manipulation of query expressions according to those rules.

The Epoq approach to extensibility results in the ability to integrate many diverse strategies for processing queries [108]. These strategies are captured in modules called *regions* which encapsulate their implementation. At the interface, then, a region (i.e. strategy) can be viewed simply as an equivalence transformation over query expressions. This observation leads to a formalization of the query optimization problem and the Epoq approach.

1.3.4 The Epoq architecture

The Epoq architecture embodies our approach to extensible query optimization. In an Epoq optimizer, query transformation is performed by a set of concurrently available region modules, each with its own strategy for manipulating query expressions. Different regions will often accomplish different query transformation tasks, but regions may also represent different strategies for accomplishing the same task in different ways. The architecture integrates these regions through a common interface for modules and a global control that combines the actions of the modules to process any given query.

The region modules are organized hierarchically, with a parent region controlling its subordinate regions as a collection of transformations. The characteristics of a query determine the order in which subordinate regions will be applied to process the query. Subordinate regions may control other subordinates, and may be arbitrary strategies for processing queries. For example, a subordinate region could be a foreign optimizer module integrated using the common Epoq interface. In this way Epoq can incorporate pre-existing optimization software with regions specifically built for the Epoq system.

This thesis presents the first definition, of which we're aware, of the control problem for extensible query optimizers (Section 5.1.3). We address the control problem with an extensible, planning-based control [105]. This control plans a sequence of region executions to process a query expression. The control is a goal-directed planner that intermingles planning with the execution of query transformations, and uses execution results to direct further planning of optimizer processing. The plans are generated from descriptions in a rule-based language of our own design. Rules describe heuristics for potentially good interactions between regions.

Epoq was motivated by a need to address extensibility in the design of object-oriented query optimizers, but we believe that it has more general utility. The architecture and control of Epoq increase the range over which any optimizer might be extended.

1.4 Outline of Thesis

Chapter 2 provides a background history of database query models and languages, and looks in detail at current work in optimization in extensible and object-oriented systems. The knowledgeable reader may omit this chapter, and perhaps refer back for comparisons while reading later chapters of the thesis.

We introduce Chapter 3 with a review of the ENCORE data model. We then present EQUAL and discuss how it supports the features of the ENCORE model. We conclude the chapter with a discussion of query transformation, and the presentation of a formal theory for query equivalence. Additional transformations for EQUAL are catalogued in Appendix A.

We then turn our attention to query optimization. Our approach to optimization is motivated by our identification of a variety of problems encountered when trying to transform queries. In Chapter 4 we discuss some of the problems encountered when trying to optimize queries expressed over an object-oriented model. These problems are categorized by modelling feature. Although the problems presented are incurred through the use of the ENCORE model and EQUAL algebra, most of them apply to other database models and languages supporting the same features.

In Chapter 5 we introduce Epoq, our strategy extensible approach to query optimization. We compare our approach to other approaches to extensibility and give a detailed example of a query processed using the Epoq approach. We give a formal basis for our approach in Section 5.4.

The Epoq architecture is described in detail in Chapter 6. We give an architecture for regions that includes specifications for a region interface and control. A large portion of this chapter is dedicated to the problem of controlling the execution of the optimizer. We present a control architecture, and describe a specific control based on planning. We describe a rule language for specifying plans, and an execution model for that language.

In Chapter 7 we present an extensible, annotated representation for query expressions that can be used by an Epoq optimizer. We describe the representation, present a structure for annotating the representation, and suggest some annotations that are useful for describing the semantics of EQUAL queries. We give an execution model for the representation, and present algorithms for building and annotating the representation.

The Epoq architecture is illustrated with the development of an example optimizer in Chapter 8. We give a design for a simple optimizer that can combine a number of different strategies to rewrite EQUAL queries. We illustrate the extensibility of our approach by adding two new strategies to the optimizer.

Finally, in Chapter 9 we summarize the scope and contributions of this thesis, and discuss a number of ideas for future research that will complement and expand on the research presented here.

Chapter 2

Query Processing: Some Background

Object-oriented database models have aspects in common with relational and network database models, complex object models, and semantic data models. In this chapter we present an overview of database models and query languages, and review optimization results for these models. In Section 2.1 we review the relational model and languages, and complex object models and languages. We then look in detail at the models for some object-oriented database systems. This gives us background for discussing the ENCORE model and EQUAL query algebra in Chapter 3.

Much of the current work in query optimization for object-oriented databases is based on optimization results for relational and extensible databases. We review query optimization for relational systems in Section 2.2, looking in particular at System R [7] and Ingres [139] as canonical examples.

The Epoq approach to optimization, presented in Chapters 5 and 6, is a new approach to extensible query optimization for an object-oriented system. Thus, it is useful to review current research in extensible, as well as object-oriented, optimization. In Section 2.3 we look in detail at some extensible systems and optimizers. Then, in Section 2.4 we examine current techniques for optimization in object-oriented systems and the optimizers proposed for some of these systems.

The survey in this chapter is brief and is designed to point out aspects of previous research that might affect the reader's understanding of the research in this thesis. A comprehensive survey of query optimization in centralized, relational databases can be found in Jarke and Koch [74]. Additional information on research in relational, extensible, and object-oriented optimization can be found in Graefe's 1989 optimization workshop proceedings [55] and in parts of his survey on optimization techniques for large databases [57].

2.1 Database Models and Languages

A database model is essentially a type system for databases. It provides the components for modelling database applications. As a result, the database model determines how an application can be logically presented.

2.1.1 The Relational Model

The relational data model is based on the mathematical notion of relations [33]. The (normalized) relations in the database are finite sets of tuples of the same kind, where each tuple contains primitive values. This provides a simple, uniform model of data which facilitates optimization, but also limits the modelling ability. The relationships that can be expressed in the model are limited to the domains themselves, relating tuples by putting them in the same relation, and relating values

by putting them in the same tuple. Most of the semantics of a database are captured in the data values. For example, the existence of tuples in a relation can be restricted by keys – values for a field or fields of tuples that cannot be duplicated in the relation. Also, when two columns in different relations are drawn from the same domain, the values in those columns can be used to compute relationships.

Codd defined an algebra and calculus (first order logic) for querying relations, and showed that they are equivalent [33]. These languages query over relations, and return relations. Details can be found in any elementary database textbook (e.g. [37], [149]).

The relational calculus forms the basis for declarative user-level query languages such as SQL [7] and QUEL [139]. These languages are declarative in the sense that the result of a query is described, but the operations that must be executed to achieve the result are not made explicit by the query. For example, the SQL-like query "**Select** p.name **From** p in People **Where** p.age > 15" describes the result as a relation of names of people over age 15, but does not give a procedure for achieving this result. Indeed, a procedure for achieving the result is what is produced by a query optimizer.

User-level relational languages are usually extended with operations for data manipulation (e.g., update, delete, aggregate manipulation). Relational languages have also been extended with the ability to apply transitive closure, but we shall not consider such languages here. For a survey of the power of such extensions to the relational algebra see [26].

The requirement that relations be normalized (i.e. tuple field values be primitive) is eased in the non-first normal form relational model [47, 63, 100, 114, 124]. This model provides more natural modelling of one to many relationships by allowing fields of tuples to be relations. Algebras for this model include the standard relational operators as well as operations for the resulting nested structures. For example, Jaeschke and Schek [73] define the operators Nest and UnNest, to change normalized relations to nested relations and vice versa. Variations on this model include the addition of ordered lists as a type [35] and requirements for tuple keys [2, 127].

More recently, extensions to the relational model include the ability to define abstract data types [67, 128, 140]. In extended relational systems fields of tuples may contain objects with abstract types. The abstract types are primitive types in the system thus, although they provide a richer modelling capability for fields of a tuple, the semantics of the database are still based on sets, tuples and the values of the typed fields.

2.1.2 Complex Object Models

Complex object models further extend non-first normal form relational models by allowing combinations of sets and tuples in any order. Complex object systems sometimes support additional types, such as lists or arrays [151], and some models allow the definition of recursive types by supporting object identity [5].

A number of languages have been proposed for complex object models (e.g. [1], [5], [11], [69], [103], [151]). Most languages are similar in power. One difference in power is whether or not a language defines a Powerset[1] operation. Languages with Powerset are strictly more powerful than the same language without the operation [1]. Another difference is the treatment (when applicable to the model), of object identity. Abiteboul and Kanellakis, for example, can produce recursive objects as the result of a query [5].

Languages for complex object models require, at least, operations for manipulation of the complex structures. For example, a Set_Collapse operator (such as [1] or [151]) takes a set (of sets) and returns the union of the set members. A similar operation can be defined for sets of

[1] Powerset creates a set of all subsets of the input set.

single-column tuples [4]. Other kinds of manipulations include building sets and tuples from single objects, grouping elements of a set into equivalence classes, and eliminating duplicates.

The definition of Cartesian product (or join) is somewhat complicated by the complex structure of objects. The product can be defined, similarly to the relational algebra operation, over sets of tuples (e.g. [4]). An alternative definition for product builds tuples containing one field for each set involved in the product (e.g. [1]). This allows for the product to be defined over sets of any type of object.

The discussion here has been brief, and has highlighted only those features of complex object models that are useful for comparison with object-oriented models. For a more complete survey of models and languages for complex objects see [4] and [70].

2.1.3 O-O Models

Object-oriented database models incorporate features of relational and complex object models as well as semantic data models (e.g. [3], [65], [71], [136]) and object-oriented programming languages (e.g. [53], [95]). Although there is no consensus on the definition of an object-oriented database, most agree that such databases must support modelling features including complex structures, abstract data types, encapsulation, object identity, subtyping, and late-binding of methods [8]. Abstract data types and encapsulation provide the ability to model the behavior as well as the structure of data. These abstraction capabilities also separate the logical view of the data from the implementation of this view.

Subtyping is a relationship between types; when two types are related through subtyping, the structure and behavior of objects of those types are also related. As a result, objects of one type can be treated, in some cases, as objects of a related type. This induces the requirement for late-binding of methods; a method defined for a type may be shared or redefined in related types, and thus the exact method to be used is not known until the type of an object is determined at execution time.

In an object-oriented database model, an object has an identity that is independent of its state. This identity is similar to a key, in that it uniquely identifies an object. It differs from a key in that it is not a value that is accessible to a database user, but at the logical level is a conceptual value that is maintained by the database system. Some extended relational models simulate this identity by defining fields containing unique identifiers (often called oids) or keys [88, 125].

Object identity allows sharing of objects; this sharing is maintained by the database and can be denoted and queried by a database user. For example, in an object-oriented database if two people work in the same department they can actually share a department object. In other database systems such sharing is accomplished by assigning an id to the department and assigning both people the same department id.

Most object-oriented database systems support both values and objects with identity. Values are immutable and cannot be shared. For example, integers, real numbers and strings are all values. An integer cannot be modified, although the memory where an integer is stored can be modified by assigning a new integer. One problem that arises in systems supporting both objects and values is the definition of types for the different kinds of entities. For example, set values are immutable collections while set objects are mutable. Mixing immutable values and mutable objects is a current area of research [39, 94].

The relationships that can be defined in an object-oriented model differ depending on the type system of the model. All models allow the definition of component relationships; an object or set of objects is defined as being part of another object, e.g., an engine is part of a car. Such relationships may be also modelled as ownership relationships – making implications about existence (an engine would not exist if the car did not exist). Many models allow the definition of classification

relationships. For example, a set of Blue Cars is a classification of the set of Cars. Interesting comparison points between models are the relationships that are primitive to the model and the ability (or lack of it) to describe new kinds of relationships. ENCORE, for example, provides a few primitive relationships with the ability to define arbitrary new kinds of relationships.

Another comparison point for object-oriented database systems is the support for sets, or other such collections. A system may explicitly, or implicitly, maintain extents for all defined types, or may require sets to be explicitly declared or built. Sets may or may not be objects (i.e. shareable), and may or may not be automatically persistent. In some systems, an extent will contain only members of the exact type, while in others extents may also include members of subtypes. The choices made will affect the way queries can be expressed.

Although most object-oriented databases contain, in name, many of the same features, they differ in the ways the features are defined, the way they are combined, and in the strength of support for different features. In the remainder of this section we quickly review the data models and languages for GemStone [98, 99], O_2 [9] and Orion [12]. These models give us a point of comparison for our discussion in Chapter 3 of the ENCORE model, as well as background information for our discussion in Chapter 4 of problems related to query optimization.

GemStone

GemStone [98, 99], developed at Servio Logic, is an early (and still current) example of an object-oriented database system implementation. The system is based on Smalltalk-80 [53], and adds typing and support for queries to the Smalltalk data model. Objects in GemStone have unique identities, respond to messages, and have a structure defined by a set of typed instance variables. Objects with the same structure and methods are grouped in classes. The classes capture the behavior of like objects, and are said to *encapsulate* that behavior. Although classes are similar to abstract data types, they do not provide data abstraction since the instance variables expose the physical structure of objects.

The classes are organized in a class hierarchy. Each object is an instance of a single class, but may also respond to messages defined for a superclass.

The OPAL language is a computationally complete database programming language that also provides the data definition and manipulation capabilities for the database system. A query is a **select:** message that can be sent to a set object along with a single argument that describes the selection condition. Path indexes defined over collections of objects can be used to provide more efficient query access [97].

O_2

The O_2 model supports two kinds of entities: values and objects. Complex structures are defined as *types* and abstract data types are defined as *classes* (see [77], [92], and [93]). Instances of a type are values in the database and instances of classes are objects with identity.

Classes have a type, which describes the structure of their instances, and associated methods which define manipulations that can be performed on objects of the class. The structure and method implementations may or may not be hidden at the logical level. A class hierarchy describes subtyping relationships between classes.

Extents for types or classes must be explicitly declared and named, and any named entity may be queried. A declarative query language [10], query algebra [32], and rule-based query language [5] have been defined for this model. The declarative language and algebra work with both objects

and values, and build only values as query results. The rule-based language, on the other hand, is unique in its use and manipulation of object identities.

Orion

In the Orion model [84] a class is a collection of objects sharing the same attributes and methods. The values of attributes of an object make up the object's state, and can be themselves objects, leading to the ability to build complex structures. The component relationships are defined in a *class-composition hierarchy* in this model. This hierarchy is essentially a description of complex tree structures involving objects with identity. A subtyping hierarchy can also be defined with subclasses derived from superclasses through the inheritance of attributes and methods. The component and subtyping relationships are the only two object relationships defined in this model. A declarative query language, which allows queries over a single class and the class-composition hierarchy for that class, has been implemented for the model [13].

2.2 Relational Query Optimization

Query optimization was important to the success of the relational model, and much of the current work in query optimization for extensible and object-oriented systems is based on relational query processing results. Two of the most well-known query optimizers are the early optimizers for System R and INGRES, so we examine these optimizers in this section.

2.2.1 System R

System R is a relational database prototype developed at IBM San Jose Research Lab in the mid-1970's [7]. The SEQUEL language is provided in the Relational Data Interface and a query optimizer for that language is contained in the Relational Data System. The optimizer contains a variety of strategies for manipulating SEQUEL statements to find a low-cost means to access data using access paths provided by a Relational Storage System.

The optimizer statically compiles SQL queries into a plan for query execution [130]. The optimizer first uses a catalog to find statistical information about relations and access paths, then determines an evaluation order for the blocks in a query. For each block, the optimizer determines join orders and access paths for relations that will result in a low-cost means of executing the block, where cost is based on disk page accesses and CPU instruction cost (comparisons). The result of optimization is a parse tree representing the chosen query solution and a plan for executing the statement.

2.2.2 INGRES

The INGRES relational database system and QUEL query language were developed, also in the mid-70s, at the University of Berkeley [139]. The query optimizer for the university version of INGRES was based on the decomposition of multi-variable queries into single-variable queries [152]. This is done interactively with query execution using two basic steps; detachment and tuple substitution.

In the detachment step the query predicates are examined to find smaller queries involving one or two relations. These smaller queries are detached from the original query to form a series of one and two-variable queries. This sequence is further decomposed by substituting all tuples from one relation into one of the two-variable queries to form a number of new queries equal to the number of tuples in the relation. This substitution choice is based on the estimated cost of the resulting

query and can take into consideration access paths for relations. It may also involve processing of some of the single-variable queries. After tuple substitution, the two steps may be repeated until the query is solved.

2.2.3 Summary

Both of these relational optimizers are customized to suit the particular database system and language used in the system. Most of the optimization strategies employed focus on queries involving the Select, Project and Join operators (see [149] for a discussion of the theory upon which such strategies are based). Subsequent work in relational optimization looked at extending optimization to include other operators, for example aggregators [38, 83]. Optimization results have also been extended to include more expressive models, such as nested [86, 126] and network models [118, 123].

The relational optimizers make extensive use of heuristics based on logical query transformation rules, such as commutativity of join and pushing select past join. A rule-based system for optimizing relational algebra expressions was proposed by Smith and Chang [138]. They use rules and algorithms for tree transformation that are based on heuristics for the relational model. For example, they have rules to move selection and projection to the leaves of a query tree. Much of the work in query optimization in extensible and object-oriented systems is based on such rules about logical transformations in the query language.

2.3 Optimization in Extensible Systems

Extensible systems are an approach to providing more functionality than relational systems. These systems are designed to allow flexibility in designing database systems for new and different applications. The term extensible refers to the idea that the system can be extended with new processing capabilities to respond to the requirements of different systems [23]. One approach to providing this extensibility is to provide a collection of tools that can be used to generate new database systems. A more open approach is to build a database system which can be easily modified to incorporate new capabilities. In this section we look at a number of extensible systems and optimizers for those systems.

2.3.1 Foundations

The earliest example of an extensible database system is Reiss' eris system [116, 117]. The goal of eris is to support new operators at the source level and new architectures, data structures and evaluation techniques at the implementation level. Flexibility in eris is provided through a data-flow architecture called the "path model". In this model, database computation is represented as a directed data-flow graph with nodes representing operators and edges representing data paths. The leaves of the graph are accesses to the database, and cost functions are associated with each operation. Query optimization involves three stages: 1) normalization of a source expression, 2) translation of the source expression into an efficient path network, using a dynamic programming approach, and 3) transformation of the path network using local optimizations. The extensibility in this system derives from the ability to model extensions in the path model, and the generality of the optimization stages.

Many subsequent systems use rewrite rules to describe the transformations that can be performed to optimize a query expression [45, 54, 115, 137]. Freytag [48] claimed that rules are sufficient for translating a user query into a query evaluation plan and, to illustrate, showed how access paths and join orders in System R optimization could be generated using transformation

rules. He illustrated an optimizer that successively applied different sets of transformations rules to manipulate a relational input query. For example, the first rules translate a query expression into an algebraic form. The next sets of rules generate access paths, join orderings, and join methods, in that order.

Freytag also outlined a generator for an optimizer. The proposed generator works in two stages. In the first stage transformation rules and internal data structures for query representation are translated into procedures (in C, for example) that transform query expressions. These procedures are input, along with a strategy for searching for applicable transformations and a cost function, in a "Combiner" phase which produces a final optimizer. Detailed information about the generator, or of resulting optimizers, has not been specified however.

2.3.2 EXODUS and Volcano

The EXODUS database system is a toolkit that allows the specification of a database model and creation of a complex object database [24]. The toolkit includes an optimizer generator [54, 58] that builds an optimizer for a specified data model. Information describing the data model, query operations, access methods, costs, and applicable transformations are provided to the generator and a rule-based optimizer is produced.

Rules in an optimizer are query rewrite rules describing transformations of algebraic operations, or implementation rules describing access methods that can be used to implement a query operation. A generated optimizer applies the given rules to a query tree to find a low-cost sequence of operations for query execution. The optimizer uses a best-first search strategy for rule-application based on the expected benefit of applying a rule to a query. The set of applicable rules is restricted by a *hill climbing* strategy; rules that may generate a higher-cost plan are considered, within some limits, just in case they might lead to further transformations that improve the plan. All generated optimizers have the same control strategy, and differ only in the application model supported and the transformation rules for that model. Extensibility is supported through regeneration of an optimizer with modifications to the input description information.

The Volcano optimizer generator improves the EXODUS generator through its use of heuristics and semantics to guide the search for transformations, its ability to learn optimization heuristics, its extensible support for physical properties of data, and its support for flexible cost models that can be used to generate plans for partially specified queries [61]. As in EXODUS, query processing is based on algebraic transformation. However, the internal representation for queries, and the search strategies used to find transformations, are more efficient in Volcano. Volcano also includes new support for operations that encapsulate parallel query execution, thus separating the optimized parallel query execution from the optimizer architecture [59].

2.3.3 Starburst

Starburst is an extensible relational database management system that provides modelling extensibility through the ability to add data types and operations for those types, storage and access methods, and internal processing methods (such as query transformation rules) [64, 128]. The query processing portion of the system works on a query represented as a graph, and processes that query by first parsing the query, rewriting the query using transformation rules, then planning an evaluation strategy for the query [64, 115].

The query rewrite stage applies rules with conditions and actions to a query graph [115]. These rules are written in a procedural language, and can be partitioned into rule sets. The rule engine can process a set of rules sequentially or in priority order.

In the execution planning stage, database access operators are constructed from low-level operators using grammar-like production rules [96]. The construction of these operators provides extensibility in the strategies that can be produced for accessing the database.

2.3.4 POSTGRES

POSTGRES [125, 142, 143] extends the relational model with the addition of abstract data types. Fields of a relation can contain objects having abstract data types, but the only subtyping defined in the model is between relations. Extensibility in the model is achieved through the ability to add new data types to describe fields of a relation as well as new access methods for those types. The new types and methods are defined by the user to be similar to types and methods that are already known by the optimizer, and are treated by the optimizer in the same manner as the known types. Optimization of queries in this model assumes knowledge about the representation of an abstract data type and requires providing selectivity and cost information for operators [140].

2.3.5 GENESIS

GENESIS differs from the previously described systems in that it is designed to support storage architectures for specialized applications, and not necessarily extensibility in the data model [14]. The system allows a database system to be customized by composing a set of previously defined modules into a new database system. The extensibility in the system is provided by the ability to pick and choose modules for each layer of the system and to define new modules for different system layers. This architecture requires that a query processing algorithm be decomposed into layers, each of which performs some kind of transformation of a query.

2.3.6 Summary

The extensible systems presented here are designed to respond to changes in the data types, operations, and access methods that are supported in a database system. In general, extensibility in optimizers for these systems is supported by the ability to describe new operations to the optimizer. These operations can be similar to existing operations and thus be supported by the optimizer's existing processing, or may require additions to the optimizer's processing. Such additions are usually supported by the use of transformation rules by an optimizer, and the addition of new rules to manipulate the new operations. The kinds of rules used to describe query transformations, and the control over execution of those rules, differ in all systems [45, 54, 115, 137].

Some systems also recognize a need to support new strategies for optimization; i.e. extensibility of the optimization process itself. This is the motivation behind Epoq, presented in Chapter 5. Other approaches to providing this same kind of extensibility [89, 129] are discussed in Section 5.1 where they can be compared with our approach.

2.4 Object-Oriented Query Optimization

Optimization techniques and results in relational and extensible databases form the basis for current research in object-oriented query optimization. In this section we first look at some particular techniques proposed for dealing with individual problems in o-o query optimization. We then look at some object-oriented query optimization systems. In particular, we will examine the optimization approach of Cluet and Delobel [29, 31], an architecture proposed by Kemper, Moerkotte and Peithner [81], and the query processing system of Straube and Özsu [144, 146].

2.4.1 Optimization Techniques

Much of the work in o-o query optimization techniques is devoted to finding efficient ways to access information referred to by a path expression. Path expressions imply a navigation from one object, through others, to a result. Research has explored the definition of indexes to speed the access through a path, clustering techniques that can be used to reduce storage accesses to retrieve portions of paths, and manipulation of the path expression itself to find alternatives to navigation.

Indexing

An early example of the use of indexing is in the Gemstone system [99]. In this system, paths follow a sequence of instance variables which are part of the structure (i.e. not behavior) of objects. Indexes are defined for each link in a path, and can be based on object identifiers or on values of the instance variables [97]. Indexes are implemented in B^+ trees, and lookup of a path involves search through a sequence of trees.

Bertino and Kim [22] call the GemStone style index a multi-index. Their work describes two other kinds of indexes for paths. A nested index uses a single index entry to denote the entire length of a path. As a result, one index access can find the beginning of a path given its endpoint. This is only useful when path uses can be predicted, though, and can be expensive to maintain. An alternative index method, the path index, associates the end of a path with all suffixes of the path in a single index entry. As a result, index manipulations involving subpaths can be performed.

The access support relations of Kemper and Moerkotte [79] are relations that store information that is equivalent to path indexes. Each column in a tuple represents a step in a path, and fields of a tuple contain object identifiers or data values. Using relations to store the paths generalizes path indexes to allow paths to traverse through sets. In addition, rewrite rules for relations can be used to manipulate the indexes during query optimization [80].

The ObjectStore system allows the definition of indexes that are similar to path indexes [111]. They also have to contend with indexing collections that are not necessarily the extents of types.[2] Their optimizer generates code alternatives, and execution of a query determines whether an alternative that uses indexes is applicable.

For a more complete exposition and comparison of indexing techniques for object-oriented databases see Bertino's survey [19].

Clustering

Another approach that is used to speed access to paths is clustering of objects. Early work on object-based clustering strategies [68, 99] physically placed objects in clusters, or segments, according to specifications by an implementor. The implementor would use knowledge about which objects would be used together to ensure they would be stored and retrieved together.

Other work in clustering looks at grouping objects logically using information about the object's structural (component) relationships or also inheritance relationships [16, 25, 131]. Recent research also addresses the dynamic re-organization of object clusters as structural object relationships change [27]. Related research dynamically determines prefetch units using pattern access histories [113].

Cluet and Delobel [31] use clustering and index information to limit the search for equivalent queries during query rewrite. Lanzelotte et al. [91] represent clustering and index information

[2]The model of Bertino and Kim [22] assumes all collections are the extents of types. Indexing collections, as opposed to extents, is also discussed in GemStone [97].

in their physical database model, but then only use index information in the representation and manipulation of queries over that model.

Manipulating Path Expressions

Another approach to finding efficient ways to traverse paths is to transform the path navigation into join operations. This is the approach of Kemper and Moerkotte with their access support relations discussed earlier [79], and is also the approach taken in the Adaplex optimizer [118] and in distributed Orion [75]. In the latter, a path is represented as connections between sets of objects (classes) in a query graph. These graphs are manipulated to find efficient traversals (forward and reverse), with access techniques such as indexes, or merge-joins, applied to the traversals.

In the Adaplex optimizer, query graphs represent nested loop accesses, quantifiers, some arithmetic expressions, and disjunctions, as well as a paths (i.e., function composition) [118]. A path is represented in a graph by connections (between sets) that represent different kinds of join operations (e.g., cross product, natural join, outerjoin). As a result, paths are manipulated by the optimizer along with the other query constructs.

Lanzelotte et al. [91] propose an optimization approach that considers select and join operations, as well as path expressions. They map a conceptual view of a query to a *processing tree* which represents a more physical view of the query. In the tree, leaf nodes are physical database sets (or subsets) and interior nodes represent joins that are explicit (i.e., a join predicate is given), implicit (i.e. a path from an object to an attribute of the object), or implicit with a path index. Their optimizer manipulates the processing tree by applying tree transformation rules. These transformations can be applied using deterministic search strategies based on the cost of the query. The transformations can result in path traversals that can start from any point in the path (not just endpoints) and can be interleaved with other query operations. This work is extended to handle recursive queries in [90].

Algebraic Rewrite

A number of proposals have been made for o-o optimization based on the algebraic rewrite of query expressions [15, 45, 49, 80, 112, 145, 155]. For example, Beeri and Kornatzky [15] provide an extensive set of rules for bulk data types that generalize existing axioms for object-oriented algebras, and assume the use of an extensible rule-based optimizer for processing the rules on a query.

Finance and Gardarin [45] present a rule language for specifying query rewrites that allows writing rules describing both syntactic and semantic transformations. They also have a meta-rule language that allows an implementor to define blocks of rules, iteration over blocks, and sequences of rules. The meta-rules help define which rules should be applied at any point, and therefore simplify the rule search strategy.

Optimizing with Methods

A recognized difficulty in applying query transformations is the problem of manipulating expressions containing references to arbitrary methods. Systems in which methods are written in the query language (e.g., [142]) can optimize the method code as a nested query. However, query languages in object-oriented databases can access arbitrary methods defined for abstract data types; these methods may be written in languages not recognized by a query optimizer. The costs associated with the application of arbitrary methods need to be considered when manipulating query expressions to determine database access strategies.

Graefe and Ward [62] address this problem by statically generating query evaluation plans with alternatives. Information available about objects is used at execution time to choose among the alternatives and generate a final evaluation plan. More recently, Ioannidis et al. address the problem of parametric query optimization [72].

The Revelation architecture describes an optimizer in which methods "reveal" information about their execution [36, 60]. The revealed information is used to expand the nodes of a query tree with execution information. The query tree, when fully expanded, could be input to a rule-based optimizer such as one generated by Volcano [61].

Bertino proposes to precompute the results of methods, and store those results using an index [20]. A method over an object O can be written in an arbitrary language, but cannot have input parameters (other than O), cannot have side-effects, and must only use primitive properties of O (i.e., properties whose values are stored as part of O). These requirements are placed to allow the system to detect when a precomputed method is invalidated. Precomputed results, when still valid, are retrieved using the index. Invalidated results require the method be computed at query execution time.

A similar approach is taken by Kemper, Kilger and Moerkotte [78]. In this system, precomputed function results are stored in relations (called *materialized functions*) along with validity flags for the data, and information about the objects contributing to a function result is maintained. The properties of types, encapsulation and object identity are used to minimize the amount of recalculation that is done when objects are updated. This approach to update management allows the system to handle arbitrary functions.

2.4.2 Some O-O Optimization Systems

Cluet and Delobel

Cluet and Delobel propose a formalism that provides for the integration of type-based path processing (e.g. [75]) with algebraic query rewrite [29, 31]. They present a model in which type information is added to a query to provide additional information for rewrite rules. The typed query can be represented as a query graph in which nodes represent sets and variables referenced in the query and are annotated with type information, and directed arcs represent operations. This graph is augmented with clustering and index information, and complemented by graphs of join and selection predicates for the query. The result is a model that integrates rewrite techniques for paths, algebraic query rewrite and common subexpression factorization, and also uses clustering and index information to reduce the search space for query transformations.

Straube and Özsu

Straube and Özsu apply relational techniques to produce a methodology for object-oriented query processing [144, 146]. Their system takes a declarative o-o query to an access plan through the following series of steps: calculus-based optimization, the transformation of calculus-based queries to algebraic form, type-checking of algebraic expressions, the application of algebraic transformation axioms, and the generation of access plans for the algebraic operations. They define a query calculus and an algebra that have only *object-preserving* operations; they cannot create new objects in response to queries.

The type-checking phase of optimization determines type consistency when query results are input to further queries. The optimizer can infer the types that result from a query application in order to ensure that further queries are well-typed.

They define syntactic and semantic rewrite rules for the object algebra which can be applied by a rule-based transformation system. The application of these rules uses heuristics to produce a query expression that can be used to generate an access plan.

Blackboard Architecture

Kemper, Moerkotte and Peithner recently proposed a blackboard architecture that presents a novel approach to the generation of access plans in a query optimizer [81]. The architecture is designed to be extensible, specializable, predictable, and tunable. Their systems generates an optimized query by processing a query in steps, where each step adds a piece to the plan.

The optimizer is organized as a ladder of *regions* on a blackboard. Between each successive pair of regions a *knowledge source* manipulates a query to move it from the lower region to the next higher region. There may be more than one knowledge source between any two regions, and any knowledge source can choose any query in the lower region to process. The processing of a knowledge source moves a query one region, but can access any of the information on the blackboard to provide additional information for its processing. All items in all regions have cost information which is used to assist the knowledge sources in their task.

Using this architecture, a query plan is generated in a fixed sequence of steps (from region 0 to region n) which can only be altered by a knowledge source that can move a query to a lower region without altering it.[3] The extensibility in this optimizer comes from the ability to add additional knowledge sources between any two regions (although the control that decides between them is not specified) and the ability to add new regions. The latter extensibility would require a complete reworking of the optimization process though, since each region represents a different query form.

2.5 Summary

Object-oriented database models have aspects in common with relational and network database models, extensible data models, complex object models, and semantic data models. Much of the current work in query optimization in the object-oriented model has been based on optimization results for relational and extensible databases.

In this chapter we reviewed these models and some of their languages, and discussed work in relational, extensible, and object-oriented query optimization. Relational query optimizers are customized for a particular system and can exploit the fixed structure and semantics of the relational system. Extensible optimizers try to generalize the optimization process to respond to changes in the data types, operations, and access methods that are supported in a database system.

Much of the research in object-oriented query optimization addresses single problems that are specific to object-oriented databases. We looked in particular at approaches to finding efficient ways to traverse paths, and approaches to dealing with arbitrary methods in queries. We also reviewed some of the optimizers proposed for object-oriented systems.

In general, the approaches and optimizers reviewed in this chapter are proposed to deal with a single problem or work in a single, fixed way to solve more general problems. However, the different problems that are encountered, and different approaches to solving these problems, indicate a need for a query optimizer that can integrate many different optimization strategies. This is the motivation behind the Epoq approach to query optimization presented in this thesis.

[3] The ability to move a query to a lower-numbered region gives iteration in plan generation.

Chapter 3

A Data Model and Query Algebra

A data model and query algebra are major components of the query processing framework described in Chapter 1. The features of the model must be supported by all components of the framework, and the characteristics of the algebra determine the transformations that can be performed on query expressions.

In this chapter we review the ENCORE object-oriented data model, and present the EQUAL query algebra for that model.[1] The data model [157] is an early object-oriented database model based on persistent objects described by abstract data types. EQUAL is a query algebra designed to support the modelling features of ENCORE. As a result, EQUAL is the first query algebra for object-oriented database systems to be completely consistent with data abstraction, and one of the few to propose operations for the creation and manipulation of objects with identity.

The ENCORE model incorporates, through data abstraction and object identity, fundamental characteristics of most other object-oriented database models. Thus, results for this model, and other components of the query processing system based on the model, could be applied to other object-oriented database models. The EQUAL algebra, through its support for the ENCORE features, is also general enough to be used as a target language for high-level, non-recursive, object-oriented languages (e.g., [29]).

In the next section we review the ENCORE data model and discuss, in particular, support for abstraction and object identity. The EQUAL algebra is presented in Section 3.2, along with some examples of its use. A major reason for incorporating an algebra, as opposed to a more declarative language, into a query processing system is that algebras are generally conducive to optimizations to find equivalent expressions that can be more efficiently executed. In Section 3.3 we discuss transformation of EQUAL expressions and present a theory for equivalence of transformed results.

3.1 The ENCORE Data Model

The ENCORE model is based primarily on data abstraction. An ENCORE type is an atomic type (Int, String, Boolean, etc.) or an abstract data type. An instance of a type is an *object*.

Abstract data types describe an interface and an implementation for instances of the type [95]. The interface is a logical description of access to instances of the type; it describes methods that can be applied to objects. The implementation describes a physical representation for instances of the type and includes implementations for methods described by the interface.

[1] Originally, ENCORE stood for Extensible and Natural Common Object REsource. After years of use, though, the name is no longer considered to be an acronym. EQUAL is the Encore QUery ALgebra.

Abstract data types and encapsulation are synonymous in this model. Objects are encapsulated since all access is through the interface, and the representation of the data and implementation of the methods is hidden. A query is concerned with the interface of a type. However, processing of a query may need to consider the implementation.

Every object in ENCORE has a unique identity that is independent of the state of the object. An object may also have a name that can be used by an application to reference the object. For example, a set of objects of type **Person** might be assigned the name *People*.

Object identity gives the ability to implement shared reference. For example, suppose a method M_x applied to an object x returns an object of type T, and method M_y applied to object y returns an object of the same type. The two methods could return the same object, indicating that x and y share the object through their M methods.

It is important to note that the methods return objects, and cannot return object identifiers. Although object identity could be implemented by a system-supplied object identifier, there are, conceptually, no identifiers in ENCORE. Methods return objects, and an ENCORE object can only be accessed through methods defined for its type.

3.1.1 Abstract Data Types and Subtypes

The ENCORE type system is based on abstract data types. Type **Type** describes abstract data types; its instances are abstract data type definitions. Types are related to each other through subtyping. Subtyping requirements ensure that instances of a subtype can be substituted as instances of a supertype (substitutability principle [153]). The subtype mechanism is designed to support strong static typing when the ENCORE type system is embedded in a compiled programming language.

All types are subtypes of type **Object**. This means that all types, and instances of types, are objects with unique identities.

The interface described by a type includes a *Name* for the type, a collection of *Supertypes* for the type, a collection of *Properties* and a collection of *Operations*. Type equivalence is by name. Any type S can have multiple supertypes T_1,...,T_n. S is said to be a subtype of any of the T_i's. (S is also a subtype of itself.) S inherits all of the operations and properties defined for each of the T_i, and can define new properties and operations. Name conflicts are resolved manually.

Properties in ENCORE reflect the abstract state of an object. *Operations* are arbitrary methods that can be applied to an object. Both are objects; i.e. they are instances of a property or operation type, respectively.

Modelling properties as objects provides a high degree of modelling accuracy, since a property can have properties of its own. For example, suppose a type Person has a "Works_for" property. The Works_for property type could define a property "Hire_date" whose value will record the date the Works_for property becomes effective for the person. This models the relationships between the person, employer and hiring date more accurately than defining Hire_date as a property of person, since the date is really a descriptor of the Works_for relationship, not of the person.

Type **Property** is the root of a subtype hierarchy for property types. It defines a special observer method *Get_Property_Value* which is inherited by all property types (i.e., its subtypes). This method, when applied to an instance of a property type returns the value of that object.

Property types are used to define relationships between objects that can be queried. All types support a method *Get_Property* that takes an object O and a property name, and returns the correct property object for O. This method, combined with the Get_Property_Value method of properties, retrieves information about the state of object relationships.

For example, if p is an object of type Person, and Person is defined as having a "Name" property,

then *Get_Property(p,Name)* returns the property object for person p. Thus, the value of the Name property for p is obtained using *Get_Property_Value(Get_Property(p,Name))*.

Property types support the definition of new kinds of relationships in the model. Type Property defines a read-only relationship. A subtype of type Property might be an updatable property, for example, with additionally a Set_Property_Value method.

Most properties we use for illustration are one-to-one relationships.[2] We abbreviate such mappings using dot notation.

NOTATION. Suppose x is an object of type T, and a property named P is defined on type T. The abbreviation x.P denotes the application Get_Property_Value(Get_Property(x, P)).

Any method f defined on a type T can be overloaded by redefining it on a subtype S. The signatures of f defined on S (denoted f_S) and f defined on T (denoted f_T) must obey the rules of contra-variance. In other words, if f_T: T,T_1,...,$T_n \rightarrow T_m$ is defined for type T (and thus has distinguished argument of type T), S is a subtype of T, and f_S: S,S_1,...,$S_n \rightarrow S_m$ is f redefined for type S (with distinguished argument of type S) then contra-variance requires that

1. $\forall i = 1, \ldots, n$ S_i is a supertype of T_i. In other words, the types of the input arguments (except the distinguished argument) to f_S must be supertypes of the input arguments to f_T. The distinguished argument to f_S will necessarily be a subtype of the corresponding argument to f_T.

2. S_m is a subtype of T_m (the output type of f_S must be a subtype of the output type of f_T).

These rules ensure that the interface of the type will obey the requirement for substitutability. Subtyping in ENCORE can be implemented using inheritance. Method implementations are bound to objects dynamically to support inheritance.

Parameterized Types

The ENCORE model supports the construction of parameterized types. These parameterized types have fixed sets of properties and operations, allowing the creation of new, strongly-typed objects. The parameterized types Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$] and Set[T] are built in, and are the only parameterized types we currently consider in the operations of the query algebra.

A parameterized type is a metatype, and the type parameters are input arguments for that metatype's Create operation. For example, the metatype Set[T] defines a Create operation that takes a type as an input argument and returns a new type as its output. If the input argument Int is provided to this Create operation, it will return a new type Set[Int]. If we call the Create operation twice, both times with the input Int, we will only create a single new type since Set[Int] is considered to be the name of the resultant type (and, recall, type equivalence is by name).

Parameterized type Set[T] defines operations for sets which include:

in (or *Member_of*) :	Set[T],T $\rightarrow$ Boolean
subset_of:	Set[T],Set[T] $\rightarrow$ Boolean

as well as all the operations in the algebra. Type T is called the *member-type* of the set. An instance of type Set[T] is a set of objects having type T or type S where S is a subtype of T. Duplication in set membership is determined using object identity; two members of a set cannot be identical.

Type Tuple associates types (T_i) with attribute names (A_i) and defines, for each attribute, methods *Get_attribute_value* and *Set_attribute_value*. The T_i's can be any database types, allowing

[2] Note though that when the destination is a set object, the relationship would be considered one-to-many.

tuples to store objects with abstract types. The value of a tuple is represented as $< A_1 : o_1, \ldots, A_n : o_n >$ where the A's are attributes of the tuple and the o's are objects of the corresponding type.

There are not necessarily subtype relationships between parameterized Set or Tuple types, since such relationships might not support the requirement for substitutability.

Properties vs. Attributes

Abstract data types in ENCORE provide the ability to logically define complex structures. Properties of an abstract data type can be viewed as attributes in that they give the ability to model the access of state information about an object. The differences between properties and attributes is important to note however. Properties in ENCORE are objects, with methods that access their values. The properties of an object define a logical structure for the object. Attributes in other systems (e.g. Orion [84], O_2 [77], or GemStone [99]) are names that have an associated stored value, thus they define a physical structure for objects. Tuple attributes in ENCORE are properties with associated methods Get_attribute_value and Set_attribute_value that can compute values associated with the attribute.

Extents

The collection of all current instances of a type is the *extent* of the type. Extents can be named and will be maintained by ENCORE when designated in the abstract data type definition. For example, in the definitions of Table 3.1 the extent of type Supplier is named "Suppliers". Extent names can be used to denote sets that can be accessed by the algebraic operations.

A subtype relationship between two types will translate into a subset relationship; i.e., the extent of the subtype is a subset of the extent of the supertype. Subsets can also be defined by predicates. The type of a predicate-defined subset is the same as the type of its superset. This gives the ability (in a limited way) to combine classification constraints with the type system, while still maintaining the ability to do static type-checking. For example, we might have the type Car with the associated extent Cars having type Set[Car]. A predicate subset BlueCars, defined as all members of the Cars set with a value of "blue" for their color property, would be a subset of Cars and would also have type Set[Car]. Although objects cannot change their type once they have been defined, they can move between sets. A given car can move from the subset BlueCars to the subset RedCars whenever its color property changes appropriately.

3.1.2 Objects: Identity and Equality

Type Object defines a family of Boolean operations we call *i-equality* (a generalization of [82]). Instances of type Object are objects with unique identities. Instances of atomic types are atomic objects that are identified by their value, i.e. the atomic object 3 has value 3. Atomic objects are identical if they have the same value (i.e. they are equal as defined by their type) and non-atomic objects are identical when they are the same object (i.e. an object can only be identical to itself).

DEFINITION 3.1. *Objects are 0-equal ($=_0$) when they are identical. For $i > 0$, two objects X and Y are i-equal ($=_i$) when they both have the same type (call it T) and*

1) if T is a collection type, then X and Y have the same cardinality and there is a one-to-one correspondence between the collections such that corresponding members are $=_{i-1}$ (i.e. there is a bijection f:X→Y such that for all x in X, $x =_{i-1} f(x)$)

or 2) if T is not a collection type then, for all properties P defined by type T, $X.P =_{i-1} Y.P$

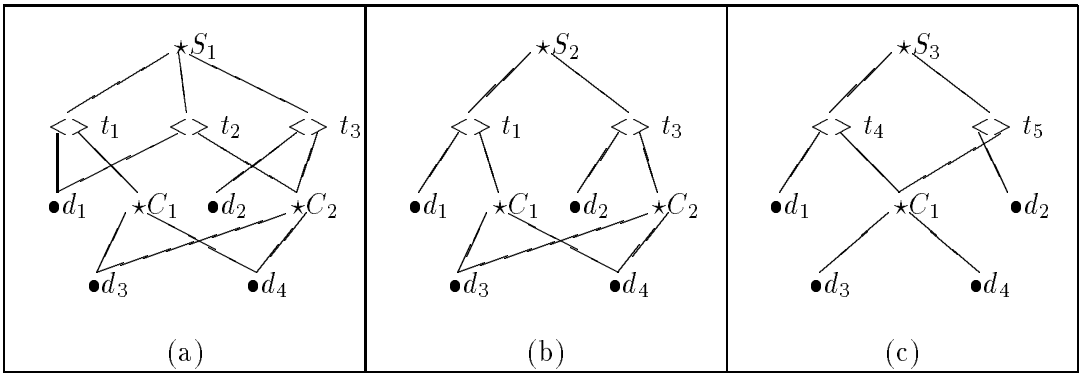

Figure 3.1: Structural representation of objects

NOTATION. The term shallow-equal refers to $=_1$; deep-equal ($=_d$) means i-equality holds for some i.

Type Object also defines a stronger notion of equality we call id-equality [133].

DEFINITION 3.2. *Two objects are id-equal at depth i if they are i-equal and graphical representations of the objects are isomorphic.*

Both kinds of equality are inherited by all types, since all types are subtypes of type Object. Abstract data types may also define type-specific equality operations; e.g., type Part may define operation Same_Part which compares two parts based on part Number and color (see Table 3.1 on page 34).

Sets and tuples are objects in ENCORE since they are instances of types. They have their own identity that is established when the instance is created. As a result, two or more sets or tuples can have the same contents; i.e., the identity of a set or tuple is not determined by its contents.

In Figure 3.1 three set objects are represented graphically. In these graphs, nodes represent objects or atomic values, arcs connect collection type objects to all objects that are members of the collection, and arcs connect non-collection type objects to the values of all properties of that object (that is, the results of Get methods for each property). Object types are labelled symbolically; $\star$ represents Set types, <> represents types for which property values will be represented, and $\bullet$ represents other abstract data types. Graphs b and c represent 3-equal set objects: C_2 and C_1 are 1-equal, making t_3 and t_5 2-equal (t_1 and t_4 are 1-equal, therefore 2-equal), thus S_2 and S_3 are 3-equal. Object S1 (graph a), on the other hand, is not related by i-equality to either S2 or S3 since it is a set with different cardinality. Also, none of the S sets are id-equal, although C_1 and C_2 are id-equal at level 1, t_1 and t_4 are id-equal at level 1, t_3 and t_5 are id-equal at level 2, and t_2 is id-equal to t_1 (and t_4) at level 2.

3.1.3 Summary

ENCORE is based on abstract data types and objects identity. These features distinguish the ENCORE model from other object-oriented database models. Other models usually combine abstract data types with structural types. As a result, they expose implementation and combine values with objects with identity. In ENCORE all objects are instances of abstract data types, and all objects have identities that are independent of their values.

The extents of types (when declared), predicate subsets, and user-defined sets are database

objects. We query over these set objects.

3.2 The EQUAL Query Algebra

The EQUAL query algebra for ENCORE provides type specific operations against collections of objects with identity. We query over collections of type Set[T] and return new objects having type Set[Q], where type Q is statically determined from the query expression.

The query algebra was designed to illustrate that operations for structurally manipulating complex objects could be adapted to manipulate objects that are instances of abstract types. Many of the operations in the algebra are syntactically similar to operations of complex object algebras. The contribution of EQUAL is the semantics supporting data abstraction and object identity (as opposed to the object identifiers that are used in complex object models).

The algebraic operations retrieve information about instances of abstract data types by requiring all access to objects in a collection to be through the interface defined for the member-type of the collection. The operations support object identity by manipulating and creating shared relationships between objects.

A query result will often be a collection of existing database objects, in which case the output member type is an existing type. However, the properties of existing types may not explicitly reflect all relationships desired by a query. Thus EQUAL provides algebraic operations that create new objects to store relationships between objects. These result objects are created using the parameterized type Tuple; objects are related to each other when they are the values of attributes in the same tuple. We also provide an operation to manipulate result tuples to indicate, when required, shared results.

Parameterized types are used for constructing new objects to ensure that the type of these objects is well-defined and can be statically checked. The type of the attributes of a tuple, or the member-type of a set, can be statically inferred from the syntax of a query in the context of the existing types.

3.2.1 The Operators

The algebraic operations can be divided into two categories:

1) operations that retrieve data: Select, Image, Project,Ojoin, Union, Intersection, and Difference.

2) operations that support data retrieval through manipulation of result structure and object identity: Flatten, Nest, UnNest, DupEliminate, and Coalesce.

The first set of operations are adaptations of operations of relational and complex object algebras to the semantics required to support our object-oriented database model. In particular, they are object-creating operations that can apply methods to database objects to retrieve information about the object properties and relationships. The Flatten, Nest, and UnNest operations are adapted from complex object algebras. The operations DupEliminate and Coalesce are included to deal with object identity and shared references.

The signatures of these operations are given in Figure 3.2. In the figure, p represents a predicate and e represents some equivalence relation (e.g. $=_i$). The operator set is not minimal. The redundancies in the operator set support logical transformations, which might be useful in query optimization. The operators are discussed in detail in the following subsections.

Select: $Set[T], p \longrightarrow Set[T]$

Image: $Set[T], f : T \rightarrow Q \longrightarrow Set[Q]$

Project: $Set[T], < (A_1, f_1 : T \rightarrow T_1), \ldots, (A_n, f_n : T \rightarrow T_n) > \longrightarrow Set[Tuple[< (A_1, T_1), \ldots, (A_n, T_n) >]]$

Ojoin: $Set[T], Set[R], A_1, A_2, p \longrightarrow Set[Tuple[< (A_1 : T), (A_2 : R) >]]$

Union,Difference, Intersection: $Set[T], Set[R] \longrightarrow Set[S]$ where S is the most specific common supertype of T and R

Flatten: $Set[Set[T]] \longrightarrow Set[T]$

Nest: $Set[Tuple[< (A_1, T_1), \ldots, (A_i, T_i), \ldots, (A_n, T_n) >]], A_i \longrightarrow Set[Tuple[< (A_1, T_1), \ldots, (A_i, Set[T_i]), \ldots, (A_n, T_n) >]]$

UnNest: $Set[Tuple[< (A_1, T_1), \ldots, (A_i, Set[T_i]), \ldots, (A_n, T_n) >]], A_i \longrightarrow Set[Tuple[< (A_1, T_1), \ldots, (A_i, T_i), \ldots, (A_n, T_n) >]]$

DupEliminate: $Set[T], e \longrightarrow Set[T]$

Coalesce: $Set[Tuple[< (A_1, T_1), \ldots, (A_i, T_i), \ldots, (A_n, T_n) >]], A_i, e \longrightarrow Set[Tuple[< (A_1, T_1), \ldots, (A_i, T_i), \ldots, (A_n, T_n) >]]$

Figure 3.2: Signatures of Algebraic Operations

Select

The *Select* operation creates a collection of database objects satisfying a selection predicate. The operation is defined as:

$$Select(S, p) = \{s \mid (s \ in \ S) \wedge p(s)\}$$

where S is a collection of objects and p is a predicate (Boolean function) defined over the type of the objects in S. If S has type Set[T], then the result is an object of type Set[T] and is a subset (restriction) of S.

The operation creates new collection objects containing all members of collection S satisfying the predicate. The predicate is any Boolean function with one input having the member type of S. In general, these functions are built as first-order predicate calculus expressions. Terms of a formula are constants (atomic objects or named objects), variables (representing objects) and functions over objects or terms. For example, the number 3 is a constant term (atomic object), the set "People" is a constant term (in this case, a named object), variable "p" is a term (representing a Person object, perhaps) and the path expression "s.Address" is a function term.[3] Terms represent objects having some type. For example, 3 has type integer, "People" has type Set[Person], "p" might have type Person, and "s.Address" has type Addr (see Table 3.1).

Formulas are formed using Boolean operations defined for the type of the terms to which the operation applies. These include *subset_of* and *in* for set types, and *i-* and *id-equality* for all types (i.e. type Object and its subtypes). For example, if s is a variable of type Supplier and p1 is a constant representing a Part object, the Boolean operation "p1 in s.Inventory" is valid since the return type of the Inventory property of variable s is Set[Part] and the *in* (member_of) predicate is defined for arguments of type T and Set[T].

Predicates are formed by combining formulas with the Boolean connectives $\wedge$, $\vee$, and $\neg$, and can be quantified by "$\exists x \ x \ in \ A$" or "$\forall x \ x \ in \ A$" where A represents a term of type Set. All terms used in a formula are bound to objects at query execution. One free variable is bound to an object in the set parameter of the Select, other free variables are bound outside the current query. For example, if variable *s* is bound by Select, and the Select is nested inside another query which binds

[3] As noted in Section 3.1.1, if Address is the name of a property of the object denoted by s, then s.Address actually denotes the composition of Get_Property_Value and Get_Property functions. Without loss of generality, we talk about this as a single function application.

a variable q, the clause $s.address = q.address$ is a valid formula in the Select predicate.[4]

The qualification "x in A" insures that x will be bound at execution time to members of an instantiated set. For example, the quantifier $\exists x\ x\ in\ Select(\ldots)$ is valid since the Select operation creates a new set object when the query is executed. The set membership qualification on quantified variables means that a predicate cannot form a Powerset operation (as in [1]). The ability to compute Powerset provides additional power that we do not want to consider at this time.

Image and Project

The *Image* operation returns a single object for each object in the queried collection, while the *Project* operation can return multiple objects for a single object. Result objects are determined by applying functions to the objects in the queried collection.

The *Image* operation has the form:

$$Image(S, f : T) = \{f(s) \mid s\ in\ S\}$$

where S is a collection of objects and f returns an object of type T. The result of the query is a collection having type Set[T]. Image returns one object for each object in the queried collection, although if f returns identical results for two different objects in a set S, the cardinality of the result set will be less than that of S. Function f must be defined for the type of objects in S and we assume f does not have any side-effects.

A function is any well-defined operation over the member type of the input set. For example, the result of operation *PartsSets := Image(Suppliers,λ s s.Inventory)* is a set of sets of parts (Set[Set[Part]]); *Image(PartsSets, λq Select(q,p))*, where p is a predicate defined for objects of type Part, is a valid operation since q represents objects of type Set[Part].

Image functions are commonly built by composing the application of properties of objects in S. When used in this way, Image is essentially the selection of objects from a collection that may not exist in the database. For example, if we assume a Supplier has an address property which, in turn, has a city property (see Table 3.1), the query *Image(Suppliers,λs s.address.city)* returns a set of objects representing the cities in which suppliers are located. This is semantically the same as $Select(AllCities, \lambda c\ \exists s\ s\ in\ Suppliers \wedge s.address.city = c)$.

The *Project* operation extends Image by allowing the application of many functions to an object, thus supporting the creation and maintenance of selected relationships between objects. The relationships are stored as tuples, with the tuple type defined by the applied functions as follows:

$$Project(S, <(A_1, f_1), ..., (A_n, f_n)>) \quad = \quad \{< A_1 : f_1(s), ..., A_n : f_n(s) > |\ s\ in\ S\}$$

S is of type Set[T], the A_i's are unique attribute names, and each f_i takes a single input of type T and returns an object of type T_i. The functions are the same as those used in the Image operation. The result type of the operation is Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$]. All properties and operations for type Tuple are thus defined for the objects of the result set.

Project returns one tuple for each object in the collection being queried. These tuples are created by the Project operation, and all have unique identifiers. The type of the collection returned is a set with all tuples uniquely identified. As a result, some tuples may contain the same values; i.e. i-equal tuples can be created. We present operations to manage such duplication at the end of this section.

[4]This assumes, of course, that the address property is defined for the types of s and q.

Project can simulate the relational Project operation by extracting only properties from the objects in the queried collection. This is similar to attribute extraction. However, the Image and Project operations are more powerful than relational projection since the function arguments could be any operations defined on the type of objects in the collection queried. This ability would be useful, for example, when updating objects in the database – an area we are not considering here.

In the Project operation, tuples can be created with components derived from different collections of objects. For example, *Project(Parts,λ p <(P,p), (J,Select(Jobs,λ j p ∈ j.PartsNeeded)>)* creates a new relationship between Part objects and the Jobs that need those objects. This type of relationship can be obtained using the Ojoin operation (which is a special case of Project; see next subsection), but is often more naturally expressed using Project. Indeed, the Project operation, when used as in this example, provides the ability to do OuterJoin [38].

Ojoin

The *Ojoin* operator is an explicit join operator used to create relationships between objects from two collections in the database. Although many relationships between objects are represented through object properties, an explicit join handles cases when a relationship is not defined in an object type. Our definition of Ojoin is designed to preserve the associativity of the operation, which is useful in defining query transformations for optimization.

Ojoin is essentially a Cartesian product of collections of objects, along with Selection of result tuples. For collections S and R, both containing objects having abstract data types, we define:

$$Ojoin(S, R, A_1, A_2, p) \quad = \quad \{< A_1 : s, A_2 : r > |\ s\ in\ S \wedge r\ in\ R \wedge p(s, r)\}$$

where p is a predicate (as in Select) defined over objects from S and R.

For example, suppose we have a set S containing stack objects and a set Q containing queues. Then $Ojoin(S, Q, A_S, A_Q, \lambda s \lambda q\ s.Top =_d q.Front)$ returns a set of stack-queue pairs in which the object at the top of the stack is deep-equal to the front element of the queue. Note that the result will be empty if S and Q do not contain objects having the same type.

The Ojoin operation creates new tuples in the database to store the generated relationships. The tuples are strongly typed using the parameterized type Tuple, with each attribute typed according to the type of the collection from which it is derived. The tuples created will have unique identities. The objects involved in the relationships are maintained and can be accessed as the value of the appropriate attribute in the tuple. In the previous example, the only operations available on the tuples are those operations defined for tuples. However, if t is a tuple in the result set, all stack operations are available for $t.A_S$ and queue operations for $t.A_Q$.

The definition of Ojoin as given does not handle sets of tuples properly, since nesting tuples within tuples destroys associativity of the operation. We thus modify the definition when sets of tuples are involved; if operand S has type Set[Tuple[$< (A_1, T_1), \ldots, < (A_n, T_n) >$]] we define

$$Ojoin(S, R, , A_R, p) \quad = \quad \{< A_1 : s.A_1, \ldots, A_n : s.A_n, A_R : r > |\ s\ in\ S \wedge r\ in\ R \wedge p(s, r)\}$$

The result is a set of (n+1)-tuples. Similarly, if R contains m-tuples, the result is a collection of (n+m)-tuples.

For example, suppose we call the previous result of stack-queue pairs SQs and join it with a set L of lists using $Ojoin(SQs, L, , A_L, \lambda t \lambda l\ t.A_S.Top = l.First)$. This result is a set of 3-tuples, where each tuple has type $< (A_S : Stack), (A_Q : Queue), (A_L : List) >$. In any tuple, the top of its stack is deep-equal to the front of its queue and identical to the first element of its list.

Set Operations

The algebra includes set operations Union, Difference, and Intersection, operation Flatten (for sets of sets) and operations Nest and UnNest (for sets of tuples). The first three operations are used to create new collections of objects, the latter three generally re-structure collections of objects.

Union, *Difference* and *Intersection* are the usual set operations with object comparisons and set membership based on object identity ($=_0$). The result for all operations is considered to be a collection of objects of type T, where T is the most specific common supertype (in the type lattice) of the types of the objects in the operands.[5] The type of the objects in a result collection is the type of, or a subtype of, the type recorded for the members of the collection. Any two sets can be combined, since all types have a common supertype of Object.

Restructuring Operations

Operation *Flatten* takes a set of sets of objects (type Set[Set[T]]) and returns a set of objects (Set[T]). *Nest* and *UnNest* extend the same operators for non-first normal form relations (see [73]) to sets of objects with identity. Sets of tuples can be unnested to convert a set-valued attribute to single-valued, or nested to create a set-valued attribute.

The *Flatten* operation is used to restructure sets of sets. We define

$$Flatten(S) = \{r \mid \exists t \; t \; in \; S \wedge r \; in \; t\}$$

where S has type Set[Set[T]], t has type Set[T] and r has type T. The result type is Set[T]; the result will not contain identical objects.

The Flatten operation is useful in conjunction with Image. Image can extract components with type Set from an object, producing a set of sets; Flatten eliminates the extra level of brackets, allowing consideration of the set members individually.

Nest and *UnNest* are non-first normal form (NF2) relational operators [73] adapted to deal with objects and identity. They both operate on Sets of Tuples, where the tuples contain arbitrary objects (in the case of Nest) or set objects (in the case of UnNest). *Nest* compares attribute values using object identity and collects values (objects) for the nesting attribute into sets:

$$\begin{aligned} Nest(S, A_i) = \{< A_1 : s.A_1, ..., A_i : t, ..., A_n : s.A_n > | \\ \forall r \exists s(r \; in \; t \wedge s \; in \; S \wedge s.A_i = r)\} \end{aligned}$$

Each of the tuples created by Nest has a unique identity, and no two tuples can be shallow-equal. However, it is possible for the A_i attribute values of different tuples to be shallow-equal. We present an operator to deal with this situation in the next subsection.

The *UnNest* operation essentially undoes the effects of Nest, although it is not an inverse operation. We define:

$$UnNest(S, A_i) = \{< A_1 : s.A_1, ..., A_i : t, ..., A_n : s.A_n > | \; s \; in \; S \wedge t \; in \; s.A_i\}$$

where the A_i attribute of the operand tuple type is set valued. As for Project, the tuples that are created have unique identities, and thus the result collection may contain shallow-equal pairs of tuples (if two tuples in S have the same values for all but the A_i attribute, for example).

Nest and UnNest are not inverse operations. If the set being nested contains shallow-equal tuples, that duplication will not be maintained by a Nest followed by an UnNest on the same

[5] Union and intersection types could be used, but then the queries would be affecting the type lattice.

attribute. For example, if we have tuple objects

$$t_1 = < A : o_1, B : o_2 > \qquad t_3 = < A : o_2, B : o_4 >$$
$$t_2 = < A : o_1, B : o_3 > \qquad t_4 = < A : o_1, B : o_2 >$$

and set $S = \{t_1, t_2, t_3, t_4\}$, Nest(S,B) creates tuples $t_5 =< A : o_1, B : \{o_2, o_3\} >$ and $t_6 =< A : o_2, B : \{o_4\} >$ and forms result $R = \{t_5, t_6\}$. UnNest(R,B) creates only three tuples – tuples shallow-equal to t_1, t_2 and t_3. Similarly, if a set R of tuples contains two tuples that are shallow-equal on all but set-valued attribute B, Nest(UnNest(R,B)) will contain fewer tuples than set R.

Object Identity and Duplication

The result of a query is a new collection of objects. Two queries cannot give identical responses, since each result collection is a newly identified object in the database. The members of a result collection may be either existing database objects or new tuple objects created during the operation. The creation of new objects means that multiple objects with unique identifiers may be created when a single object is desired. We define operations *DupEliminate* and *Coalesce* to handle situations where "equal" objects are created by a query.

Operation *DupEliminate(S,e)* provides the option of eliminating duplicate copies of objects from a collection. The parameter e is an equality operator used to determine duplication. Such elimination is automatic for identical objects (a set cannot have two identical members), but operations that can create new objects (such as Project or UnNest) may create objects that are i-equal. In Figure 3.1 (page 27) S_1 is a collection containing 2-equal objects (t_1 and t_2). Graph b of the figure represents the result of executing $DupEliminate(S_1, =_2)$.[6]

Operations such as Project and Nest can create tuples with *components* (attribute values) that are equal (usually shallow-equal). We thus define operation *Coalesce(S,A_k,e)* which, for collections S of tuple objects, eliminates e-equal duplication for the A_k attribute values of the tuples, where e is, again, any type of equality defined for the type of objects in S. If two or more tuples in S have e-equal A_k attribute values, then one of the objects is chosen to represent A_k in the result. The result tuples are copies of the tuples from S, with each representative A_k value replacing the A_k attribute value in the appropriate tuples.

For example, in Figure 3.1, Coalesce(S_2,C,$=_1$) can result in object S_3 (figure c). The set-valued attributes of tuples t_1 and t_3 in S_2 are 1-equal ($C_1 =_1 C_2$) thus the Coalesce results in identical set-valued attributes in t_4 and t_5 of S_3 ($t_4.C =_0 t_5.C$).

3.2.2 Examples

We illustrate the operation of the algebra with some example queries on a simple object-oriented version of a Supplier-Parts-Job database [37]. Types Supplier, Part, Job and Addr are defined in Table 3.1. For the purposes of these examples, we assume that Type Object is the only supertype for each of the given types. Sets Suppliers, Parts and Jobs, with the obvious associated types, will be used for the queries.

The properties of the types are used as a scheme for each collection and the type of each property determines what operations are available to the query. For example, assume Ordered_list is a user-defined type with properties Empty, Member_of, First, Last, etc. We can determine the values of any of these properties for the Preferred_Suppliers object of any object in Jobs. All set

[6]The result might instead contain tuples t_2 and t_3, since DupEliminate nondeterministically chooses a representative from the set of duplicate objects (i.e. from the equivalence class).

ADT Supplier
Extent: Suppliers
Properties:
ident: string
address: Addr
Inventory: Set[Part]
Operations:
RecvOrder:
Supplier, Set[Part] → Supplier

ADT Job
Extent: Jobs
Properties:
num: string
address: addr
PartsNeeded: Set[Part]
Preferred_Suppliers:
Ordered_list[Supplier]
Operations:
NewPart: Job, Part → Job

ADT Part
Extent: Parts
Properties:
Num: string
address: addr
color: string
components:
Set[Tuple[<(P,Part),(Qty,Int)>]]
plan: drawing
BillofMaterial: list[Part]
Operations:
Order: Part → Part
Same_Part: Part, Part → Boolean

ADT Addr
Properties:
street: string
city: string
state: string

Table 3.1: ENCORE types for a Supplier-Parts-Job database.

operations for the model (e.g., in, subset_of) as well as the set operations defined by the query algebra can be applied to objects having type Set.

EXAMPLE 3.1. *Find all red parts. Which suppliers can supply all of the red parts?*

P_red := Select(Parts,λp p.color = "Red")
S_Pred:= Select(Suppliers,λs P_red subset_of s.Inventory)

The first selection finds the red parts, the second selection finds all suppliers for which the inventory includes that set of parts. The *subset_of* operation is available since property Inventory and result P_red both have type Set[Part].

EXAMPLE 3.2. *What parts are needed by jobs in Boston?*

BosJobs := Select(Jobs,λj j.address.city = "Boston")
BosParts:= Flatten(Image(BosJobs,λj j.PartsNeeded))

The first operation selects all jobs in Boston and the Image operation extracts the PartsNeeded sets from those jobs. The Flatten operation converts the set of sets of parts into type Set[Part]. If we want to retain information about which parts are needed with which job we could use a Project operation instead of Image:

BosJobParts := Project(BosJobs,λj <(J,j), (Pt,j.PartsNeeded)>)

Note that operation NewPart (of type Job) cannot be applied to members of BosJobParts since they have type Tuple. However, if b represents a member of BosJobParts, then NewPart can be

applied to b.J. The results of the queries maintain the instances of type Job and provide multiple paths to Boston-based jobs (through collections Jobs and BosJobs, and property J of objects in BosJobParts).

EXAMPLE 3.3. *Find all local suppliers for each job.*

LocalS := Ojoin(Jobs,Suppliers,J,S,λj λs j.address.city = s.address.city)

The result is a set of tuples of type <(J,Job), (S,Supplier)>. This result is similar to a normalized relation. An application of operation $Nest(LocalS, S)$ would give a set of suppliers for each job, i.e. a collection of tuples having type <(J,Job),(S,Set[Supplier])>. A similar result can be accomplished using

Project(Jobs,λj (<(J,j),(S,Select(Suppliers, λs s.address.city =j.address.city))>))

For each job the set of suppliers in the same city is selected from the collection of all suppliers. However, if there were no local suppliers for a job this would result in tuples with null values (an empty set) for the S attribute. Although Project, with embedded Select, can simulate Ojoin (with formatting differences handled by UnNest and tuples with empty sets removed by Select), we offer both operators for user clarity as well as optimization potential.

EXAMPLE 3.4. *Find the preferred local suppliers for each job.*

We assume that type ordered_list has a Boolean operation *In_List* which can be applied to objects of that type, and select from result LocalS of the previous example.

Select(LocalS,λr r.S *In_List* r.J.Preferred_Suppliers)

The result could also be obtained in a single Ojoin operation by conjoining the selection predicate of this example (with r.S changed to s and r.J to j) with the predicate of the Ojoin in Example 3.3.

EXAMPLE 3.5. *Which parts can be supplied by every supplier?*

AllParts := Flatten(Image(Suppliers,λs s.Inventory))
PartsIntX := Select(AllParts,λp $\forall$s (s in Suppliers $\wedge$ p in s.Inventory))

AllParts is the union of the Inventory sets of all suppliers. From that union, we select those parts that are in every Inventory set. In the Select predicate, variable p is bound to objects in AllParts and variable s is bound to objects in the Suppliers set. The query could be stated without using universal quantification as follows:

PartsIntX:= Select(AllParts,λp
Empty(Difference(Suppliers, Select(Suppliers,λs p in s.Inventory))))

In general, universal and existential quantification can be eliminated from Select predicates by using nested Select operations with set Empty and (possibly) Difference operations.

Representing Relations

Querying a relational database can be modelled using EQUAL and ENCORE. Normalized relations are easily represented in our model as collections of type Set[Tuple] where all attributes of the tuple have atomic types. The relational algebraic operations map directly to our algebra applied to sets of tuples.

Conversely, the Project operation can be used to map property values of objects in our model to relations. The tuples representing objects will retain, as attribute types, the property types. These attribute types could be treated as atomic types (e.g., Part.plan) or could, in some cases, be further expanded using UnNest and Project. For example, the following query creates, from the Suppliers set, a relation containing tuples having type <(ident,string), (address,Addr), (Inventory,Set[Part])>.

NNR_Suppliers := Project(Suppliers,λs < (ident,s.ident),
(address,s.address),
(Inventory, s.Inventory)>)

The relation could be further expanded, if desired, using Project to separate the address into its component parts (street, city and state) and UnNest to create tuples where the type of Inventory is Part instead of Set[Part].

3.2.3 Summary

The support for abstract data types, object identity, type inheritance, and parameterized types by the ENCORE data model is integral to the design of the EQUAL query algebra. The distinction between types and collections in the model allows the algebra to query over collections, using a scheme for each collection described by the type structure. The algebraic operations return collections of objects, and the type of those collections can be statically determined using the type structure. Objects in a collection will have the type defined for the collection, or will have a subtype of the type defined for the collection.

Although the operations of EQUAL are syntactically similar to operations for relational and complex object algebras, the EQUAL operations are semantically quite different. The EQUAL query operations support our object-oriented database model with object-creating operations that apply methods to database objects to retrieve information about the object properties and relationships. The operations can also detect and manipulate relationships that share objects, and support object identity in a more abstract way than algebras that manipulate explicit object identifiers.

The query algebra supports abstract data types by accessing objects only through the interface defined by their type. The manufacture of tuples by the Project and Ojoin operations allows the expression of new relationships not identified by the type design. Static typing of query results is supported by the parameterized types.

The manipulation of objects by the algebraic operations is based primarily on object identity, and most operations create new objects (tuples, collections) to store query results. The algebra supports object identity by maintaining the identities of existing objects and by providing operations to manipulate, when desired, the identities of objects created to store results.

The support for object identity requires multiple notions of object equality in query operations. Identity is the default equality for the algebraic operations, but value-equality (i-equality), structural equality (id-equality) and arbitrary, user-defined equalities can be explicitly denoted in some operations. For example, object comparisons in Select or Ojoin predicates can explicitly denote any type of equality defined for the objects being compared. Operations DupEliminate and Coalesce also allow any equality operator to be designated for comparisons. However, there are many situations in the algebra when implicit comparisons are made, and those comparisons always use 0-equality (identity). For example, operation Intersection uses 0-equality to determine which objects of one set are also present in another. The algebraic operations use 0-equality to eliminate duplication in result sets (of course, this can be overridden by using DupEliminate). The Nest operation uses 0-equality when comparing attributes and when creating the nested sets. The

implications of the use of 0-equality as a default when performing operations on sets should be considered when posing database queries.

When restricted to the relational model, the EQUAL operations can simulate the relational algebra and thus would support relational systems embedded in an object-oriented database. As a relational simulation, EQUAL is a complete database language [26]. With the full expression of EQUAL on an object-oriented model, we can express first order queries and we chose to limit the algebra to not express recursive queries.

3.3 Query Transformation

In order to manipulate queries in an optimizer it is necessary to define transformations which preserve the equivalence of query expressions. In a value-oriented setting, equivalence of query expressions means equality of query results. In our object-oriented model, however, every entity in the database is an object with identity. The algebra, in turn, produces new objects as the results of queries. As a result, two responses to even the same query cannot be identical. The creation of new objects by the algebraic operations means that the logical structure of the result, as well as the data retrieved by a query, must be considered when defining equivalence.

In this section, we first look at the meaning of equivalence of query expressions in object-creating algebras such as EQUAL, and give three different definitions for query equivalence. We then look at some transformation rules for EQUAL expressions that illustrate our definitions. In Appendix A we provide an inventory of transformations that support structural equivalence (definition 3.5).

3.3.1 Equivalence

The logical structure of an object can be represented graphically (as in Figure 3.3). Let R be the result of some query Q (on some database D), and G(R) = (V(R),E(R)) be the graphical representation of object R. V(R) is the set of nodes in the graph (these represent objects) and E(R) is the set of arcs connecting elements of V(R) (representing relationships between objects). Consider two disjoint subsets of V(R):

1) V_Q(R) — the set of objects in V(R) that were created by query Q

2) V_{Qdata}(R) — the set V(R) − V_Q(R)

The set V_Q(R) will only contain objects having type Set or Tuple, since these are the only types currently created by queries. Also, V_Q(R) will always contain at least one element, since a query always returns a new set object. V_{Qdata}(R) contains, intuitively, the set of database objects returned by query Q.

One notion of query equivalence considers two queries to be equivalent if they return the same objects from the database (i.e. their results have the same Qdata sets):

DEFINITION 3.3. *Two queries, Q_1 and Q_2, are weakly equivalent ($\approx$) if, when applied to an arbitrary database, they return results R_1 and R_2, respectively, with corresponding graphical representations $G(R_1)$ and $G(R_2)$ such that $V_{Q_1 data}(R_1) = V_{Q_2 data}(R_2)$.*

Note that the Qdata sets are mathematical sets, not objects, thus equality of the Qdata sets is set equality.

Weak equivalence disregards the types of the objects containing the results, i.e., it disregards the objects built by a query to hold results. Although both result objects are sets, they could be sets of different types or different cardinalities. For example, one could be a set of tuples containing

objects of type T (e.g., Set[Tuple[A:T,B:T]]) and the other a set of objects having type T (i.e., Set[T]), where the objects of type T are retrieved from the database. These two results cannot be compared by the object equalities defined in Section 3.1.2 since they have different types.

Weak equivalence ignores the structure built by a query to store the results. Thus an application of any of the formatting types of instructions (e.g., UnNest, Nest, DupEliminate) will not affect weak equivalence.

A stronger notion of equivalence uses i-equality and requires results to have the same type:

DEFINITION 3.4. *Two queries are equivalent at depth i (or i-equivalent, represented $\cong_i$) if, when applied to an arbitrary database, they return i-equal objects.*

The equivalence depth will depend on the query. Nested queries that build tuples will increase the depth of the structure built by the query. The objects created by the query operation will have unique identities, and this definition implies an identity-oblivious search through the structures to find the data retrieved from the database. For example, any two non-set objects are compared property-wise, thus two distinct objects in one set that have identical values for a given property could match with two distinct objects in a different set with i-equal values for the same property. The fact that the two objects in the first set have aliases for the property value is lost with i-equivalence.

The transparency of object identifiers in i-equivalence means a loss of information about shared references in results. Two results that are i-equivalent may have non-isomorphic graph representations (see Figure 3.3 b and d, for example) meaning that information about aliasing of objects may be lost. Thus an even stronger notion of equivalence is based on id-equality:

DEFINITION 3.5. *Two queries are structurally equivalent at depth i (also called id-equivalent, and represented $\equiv_i$) if, when applied to an arbitrary database, they return objects that are id-equal at depth i.*

The three definitions of equivalence are exemplified in Figure 3.3. In the figure, all d_i's are database objects (they have arbitrary data types) and S_i's, t_i's and C_i's are objects created by the execution of query operations. S_1, S_2, S_3 and S_4 are result collections from four queries. All four results are weakly equivalent; they all return d_1, d_2, d_3, and d_4 from the database. However, S_1 is not i-equal to any of the other results since it is a set of 2-tuples where both tuple attributes (properties) are abstract data types. All of the other sets contain 2-tuples where one attribute is known to have type Set. Sets S_2 and S_3 are structurally equivalent at depth 3; the assignments $S_2 \leftrightarrow S_3, t_4 \leftrightarrow t_6, t_5 \leftrightarrow t_7, C_1 \leftrightarrow C_3$, and $C_2 \leftrightarrow C_4$ illustrate the isomorphism with the corresponding C's being 1-equal. S_4 is i-equivalent at depth 3 to S_2 and S_3, but not id-equivalent. In particular, in S_4 both t_8 and t_9 have identical values for their set-valued attribute (i.e. C_1) but in S_2 (for example) the corresponding set-valued attributes in t_4 and t_5 are 1-equal. In S_4, changes in $t_8.C$ can affect t_9. That is, the set-valued attribute is aliased.

The definitions of equivalence illustrate the strengths, and difficulties, of querying when objects have identity. A query not only returns data, but defines some relationships between objects in the database. The operators allow the specification of those relationships and the description of how the relationships will be stored in objects. The different definitions of equivalence allow differences in the relationships stored (weak equivalence vs. i-equivalence) and in the logical structure of the objects storing those relationships (i-equivalence vs. structural-equivalence).

3.3.2 Transformations

The similarities between EQUAL and the relational algebra, and the flexibility of the operator set in handling data types and identities, allow the definition of a number of transformations over the

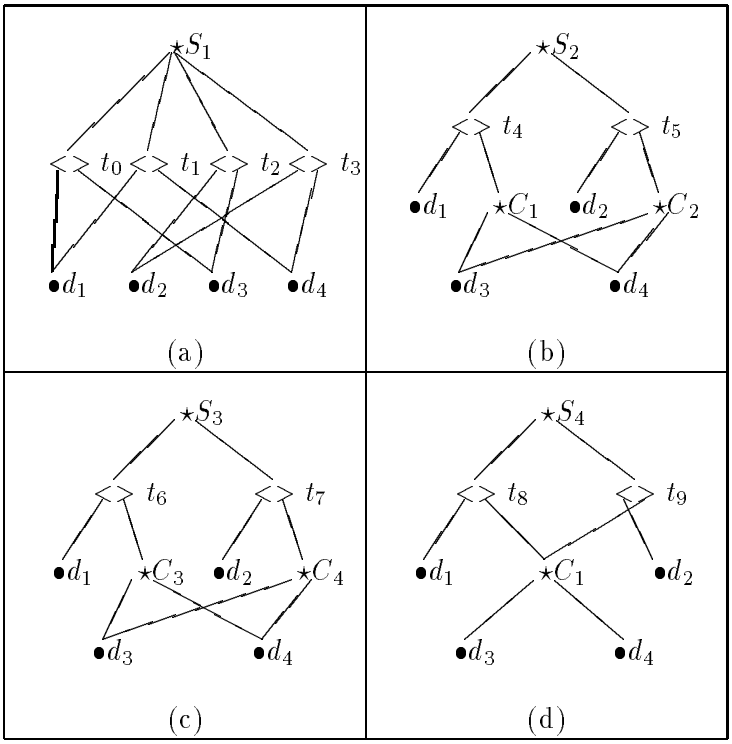

Figure 3.3: Equivalent query results. All results are weakly equivalent; S_2 and S_3 are id-equivalent to each other and i-equivalent to S_4.

algebra that produce results that are equivalent by the definitions presented in the last section. A transformation over query expressions satisfies an equivalence when the queries denoted by the expressions are equivalent. In Figures 3.4 and 3.5 some transformations satisfying the different definitions of equivalence are identified.

Identities 3.1 through 3.6 of Figure 3.4 are transformations giving weakly equivalent results. Weakly equivalent queries often return data in different formats. For example, identity 3.1 states that a Project operation can return the same data as an Ojoin, although the Project operation returns a nested tuple whereas Ojoin returns "flat" tuples. The type of a result can also be manipulated by the inclusion or exclusion of the formatting types of instructions in query expressions. The operations Flatten, Nest and UnNest modify the type, but not the data, returned by a query. Identity 3.5, for example, indicates that a query operation producing tuples with a set type attribute, followed by an UnNest, will produce a result that is weakly equivalent to the same query without the UnNest. The query results with and without the UnNest have different types.

I-equivalence maintains the type of queries, and weakly equivalent queries may have the same type. In particular, DupEliminate and Coalesce manipulate the identities of objects but not their types. Any query is weakly equivalent to the same query followed by an application of DupEliminate (identity 3.6). The result sets will contain the same types of objects, but are not i-equivalent because they may have different cardinalities. Similarly, Coalesce eliminates duplication in an object's structure and thus, when applied after any other algebraic operation, produces a result that is not id-equivalent but i-equivalent to the operation without the Coalesce (identity 3.7).

Weak equivalence does not guarantee that relationships between data objects will be maintained.

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a,b)) \approx Select(Project(As, \lambda a < (A, a), (B, Select(Bs, \lambda b\ p(a,b))) >), \lambda t\ NOT\ Empty(t.B)) \quad (3.1)$$

$$Flatten(S) \approx UnNest(Project(S, \lambda a < (A, a) >), A) \quad (3.2)$$

$$Image(As, \lambda a\ Select(Bs, \lambda b\ b.prop = a.prop)) \approx Select(Bs, \lambda b\ \exists a\ (a\ in\ As \wedge b.prop = a.prop)) \quad (3.3)$$

$$Image(As, \lambda a\ f(a)) \approx Project(As, \lambda a < (F, f(a)) >) \quad (3.4)$$

$$UnNest(NSTupleQuery, A) \approx NSTupleQuery \quad (3.5)$$

$$AnyQuery \approx DupEliminate(AnyQuery, =_i) \quad (3.6)$$

$$AnyTupleQuery \cong_{i+2} Coalesce(AnyTupleQuery, A, =_i) \quad (3.7)$$

Figure 3.4: Some query transformations.

For example, in identity 3.3 assume that As are Stacks, Bs are Queues, and *prop* is the length property for stacks and queues. The Select operation creates a collection containing all queues for which there is some stack having the same length. The Image operation returns the same queues, but they are collected into sets of queues having the same length (for each stack, Image creates one set of queues having the same length as the stack). Each set in the Image result represents a relationship (of length) between queue objects that is not explicit in the Selection result.

Operations that create new objects can permit duplication in the data returned. For example, consider identity 3.4 relating the Image and Project operations. The queries are only weakly equivalent since they return different types.[7] In addition, the Projection result may have a different cardinality than the Image result since the former may contain tuples that are 1-equal (function f could return identical objects for different input objects). Operation DupEliminate could be applied after the Project to eliminate any duplication giving, again, a weakly equivalent result. The Project and DupEliminate-Project results have the same type, but are not i-equivalent since they contain different numbers of objects.

Some id-equivalent transformations are displayed in Figure 3.5. Identities 3.8 and 3.9 in Figure 3.5 are id-equivalent versions of identity 3.1 of Figure 3.4. A Project, with Selection and UnNesting for formatting, is structurally equivalent to Ojoin. Similarly, identity 3.10 is the id-equivalent version of identity 3.2. The Image operation in the former "converts" the single-attribute tuples into single objects having the attribute type.

Id-equivalent transformations are useful in identifying redundant operators within the algebra. Identities 3.8 - 3.11 illustrate the redundancy of operators Ojoin, Flatten and Coalesce, respectively. Identity 3.12 illustrates that some predicates with quantifiers can be replaced by algebraic operations using propositional predicates. The redundancy in the operator set may offer opportunities for optimization. We would expect, for example, that a Flatten operation would be more efficient than the simulation of that operation with Project, UnNest and Image, since the Flatten operation more succinctly describes the required result.

Relational transformation results can also be applied to the object-oriented query algebra, resulting in id-equivalent transformations. For example, one relational optimization strategy is to push Selection past Join. That same strategy can be defined for EQUAL (identity 3.13). Similarly, when a Select operation is composed with an Ojoin, it may be possible to instead compose the two predicates to produce a single operation (identity 3.14). These two ideas are combined in identity 3.15 of Figure 3.5. If an Ojoin predicate contains a conjunct involving only one of the operand collections, that conjunct can be extracted from the Ojoin predicate to form a Select

[7] Set[Tuple[<F:ftype>]] vs. Set[ftype], where ftype is the return type of function f.

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a,b)) \equiv_2 Select(UnNest(Project(As, \lambda a < (A,a), (B, Bs) >), B), \lambda t\ p(t.A, t.B)) \quad (3.8)$$

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a,b)) \equiv_2 UnNest(Select(Project(As, \lambda a < (A,a), (B, Select(Bs, \lambda b\ p(a,b))) >), \lambda t\ NOT\ Empty(t.B)), B) \quad (3.9)$$

$$Flatten(S) \equiv_1 Image(UnNest(Project(S, \lambda s < (A,s) >), A), \lambda a\ a.A) \quad (3.10)$$

$$Coalesce(S, A_k, =_i) \equiv_{i+2} Project(Ojoin(S, NoDupA_k, \lambda a \lambda b\ a.A_k =_i b), \lambda t < (A_1, t.A_1), \ldots, (A_{k-1}, t_{k-1}), (A_k, t.B), (A_{k+1}, t.A_{k+1}), \ldots, (A_n, t.A_n) >) \quad (3.11)$$

$$\text{where} NoDupA_k := DupEliminate(Image(S, \lambda s\ s.A_k), =_i)$$

$$Select(As, \lambda a\ \exists b\ (b\ in\ Bs \wedge p(a,b))) \equiv_1 Image(Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a,b)), \lambda t\ t.A) \quad (3.12)$$

$$Select(Ojoin(As, Bs, A, B, p), \lambda t\ p_s(t.A)) \equiv_2 Ojoin(Select(As, p_s(a)), Bs, A, B, p) \quad (3.13)$$

$$Select(Ojoin(As, Bs, A, B, p(a,b)), \lambda t\ p_s(t.A, t.B)) \equiv_2 Ojoin(As, Bs, A, B, p(a,b) \wedge p_s(a,b)) \quad (3.14)$$

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a) \wedge p'(a,b)) \equiv_2 Ojoin(Select(As, p(a)), Bs, A, B, \lambda a \lambda b\ p'(a,b)) \quad (3.15)$$

Figure 3.5: Some structurally-equivalent query transformations.

predicate on the appropriate operand.

We expect most optimizers to use transformations that maintain structural equivalence. However, we will show in Chapter 4 that weaker notions of equivalence can be used in interim stages of a query transformation that produces, ultimately, a structurally equivalent result.

3.4 Summary

The ENCORE database model forms a strong base for the exploration of object-oriented query processing. The model is based on data abstraction and object identity – two key features of object-oriented systems. Unlike most other models, ENCORE does not mix values with objects and does not allow access at the logical level to the physical structure of objects.

The EQUAL algebra supports abstraction and identity in the ENCORE model and, at the same time, provides the ability to construct dynamic relationships. The algebra supports abstract data types by accessing database objects only through their interface. Results of queries are collections of existing objects or collections of tuples built by the query. The creation of collections and tuples with statically determined types is supported by ENCORE parameterized types.

The algebra supports object identity by managing the creation of new objects and by considering object identity when manipulating these objects. EQUAL provides operators to manipulate the structure of new objects created by a query, and to manipulate the relationships between objects indicated by identities. For example, EQUAL provides a Coalesce operator to build aliased structures.

The ability to transform algebraic expressions to find expressions that are efficient to execute makes an algebra a critical component of a query processing system. Transformation of algebraic expressions is based on the equivalence of query results. In Section 3.3.1 we gave three definitions for query equivalence – weak equivalence is based only on the database objects returned by a query, i-equivalence recognizes the preservation of relationships between database objects, and structural equivalence recognizes differences in the logical structure of objects storing query results (i.e., shared references) . In the remainder of this thesis, we will assume an optimizer transforms using

structural equivalence since this more fully supports object identity. In Appendix A we inventory transformations that maintain structural equivalence for EQUAL expressions.

Chapter 4

Problems in Object-Oriented Query Optimization

Although many of the problems that must be solved by an object-oriented query optimizer are similar to problems solved by relational and extensible optimizers, there are also many problems that are unique to the object-oriented model. In this chapter we present some problems that arise when optimizing object-oriented queries. The problems are organized by the object-oriented modelling feature which generates them. The problems are presented in the context of the ENCORE model and EQUAL algebra, but generalize to any models supporting the feature generating the problem.

In particular, we look at problems that arise as a result of supporting abstract data types, complex structures, methods and encapsulation, and object identity. We discuss each problem in the context of the particular feature with which it is associated, and note any relationships between a problem being described for the object-oriented model and problems encountered in relational optimization. For example, the nested queries common in object-oriented languages generalize nested queries in SQL. We also note proposed techniques for solving the problems, where applicable.

The examples in this chapter will refer to the sample scheme of Table 4.1. The scheme represents a car-manufacturer database (similar to [13]) in which companies have departments, and vehicles are manufactured by companies. There is also a hierarchy of people which includes employees, managers (specialized employees), and students who study at a company under the direction of a manager. Some students are paid while they study (type Student-Employee).

These are ENCORE types, so the schemes shown describe methods that can be applied to instances of a type to disclose state information (the *Properties*), the subtype lattice (*Supertypes*), and collections maintained by the database (the *Extents*). We do not show *Operations* in the scheme since we want to query without side-effects. As noted in our discussion of ENCORE, the information shown is at the interface of the abstract data type. We assume no particular implementation or physical representation for the *Properties*.

The Properties of any type in the scheme include those explicitly listed for the type as well as all properties of its supertypes. For example, type Manager has Properties budget, secretary, employer, mgr, department, salary, name, age, cars and residences. Type Student-Employee has two different properties of type Company; Property *comp* is the Company inherited from type Student and represents the company at which the student studies, and Property *employer* is inherited from type Employee and represents the company that pays the student. Type Student-Employee also multiply inherits its *mgr* property. We are not concerned with the resolution of that conflict here.

In Section 4.1 we discuss general problems that arise in supporting abstract data types and

ADT Vehicle
Extent: Vehicles
Properties:
vin: string
manu_by: Company

ADT Company
Extent: Companies
Properties:
name: string
emps: Set[Employee]
depts: Set[Dept]

ADT Dept
Extent: Departments
Properties
mgr: Manager
emps: Set[Employee]

ADT Address
Properties:
street: string
city: string
state: string
zip: string

ADT Person
Extent: People
Properties:
name: string
age: integer
cars: Set[Vehicle]
residences: Set[Address]

ADT Employee
Supertype: Person
Extent: Employees
Properties:
employer: Company
mgr: Manager
department: Dept
salary: real

ADT Manager
Supertype: Employee
Extent: Managers
Properties:
budget: real
secretary: Employee

ADT Student
Supertype: Person
Extent: Students
Properties:
gpa: real
comp: Company
mgr: Manager

ADT Student-Employee
Supertype: Student, Employee

Table 4.1: Example scheme.

in Section 4.2 we look at problems that are specific to supporting the complex logical structures that can be built using an abstract type system. In Section 4.3 we discuss the problems related to supporting encapsulated methods. Finally, in Section 4.4 we examine problems that arise when supporting object identity in an object-oriented database. In each section we look at how the systems and techniques surveyed in Chapter 2 address the different problems.

4.1 Abstract Data Types

In an object-oriented model we need to incorporate optimizations for a changing variety of types. Queries may be based on operations over collections, but optimizations pertaining to sets (or other bulk types – multisets, lists, etc.) need to be combined with optimizations over the types of the objects contained in the collection. Similarly, these optimizations can involve optimizations with the types of objects related to objects in the set (by properties, for example) and so on.

4.1.1 Type Specific Optimizations

An object-oriented query optimizer must be able to apply optimizations specific to the types, and optimizations that look at relationships between objects of different types. Such optimizations are similar to those examined in the area of semantic query optimization [85, 101, 135].

For example, suppose we have the query clause

$$\text{e.employer.name} = \text{v.manu\_by.name}$$

where e is an object of type Employee and v is a Vehicle, and suppose we can define the following axiom (analogous to a functional dependency) for abstract data type Company:

$$\forall c_1, c_2 : \text{Company} \quad c_1.\text{name} = c_2.\text{name} \Longrightarrow c_1 = c_2$$

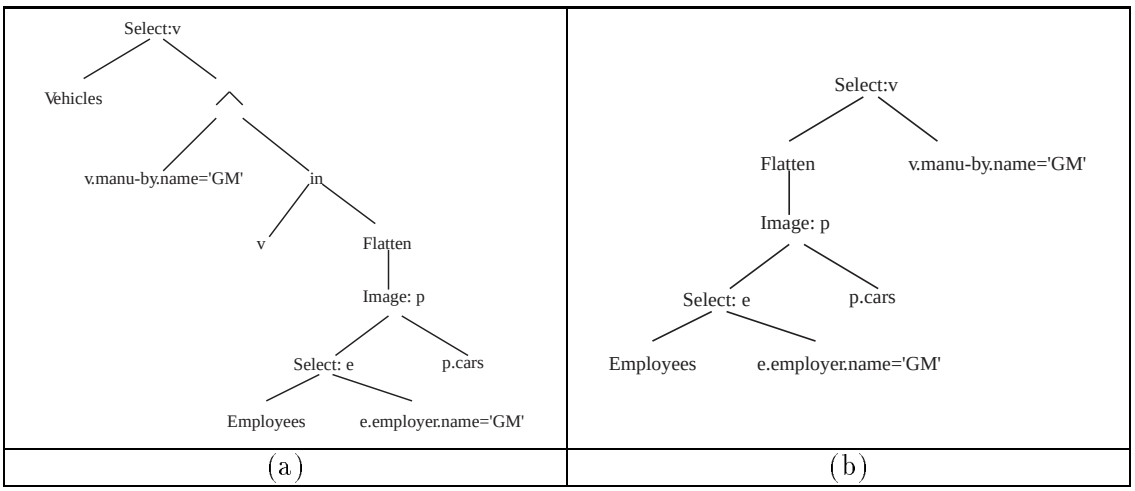

Figure 4.1: Query — Find all GM vehicles owned by GM employees.

A type-specific optimization could note that e.employer and v.manu_by both refer to Company objects, and apply the axiom to the predicate to simplify it to

$$\text{e.employer} = \text{v.manu\_by}$$

This kind of simplification depends solely on the information about the abstract data type. It would be useful to incorporate such transformations into an object-oriented query optimizer. This requires that the optimizer must be able to be extended with new transformations when new types and constraints are added to the system.

4.1.2 Subtypes and Subsets

An object-oriented query optimizer could also have opportunities to apply transformations that use knowledge about the abstract data type construct. For example, consider the query "Find all GM vehicles owned by GM employees". One way to evaluate this query is represented as the operator tree of Figure 4.1a. This evaluation corresponds to the following sequence of algebraic operations:

$$\begin{aligned}
\text{GMemps} &:= \text{Select (Employees,}\lambda\text{e e.employer.name = ``GM'')} \\
\text{GMempCars} &:= \text{Flatten(Image(GMemps,}\lambda\text{p p.cars))} \\
\text{Answer} &:= \text{Select(Vehicles,}\lambda\text{v v.manu_by.name = ``GM''} \\
&\qquad\qquad \wedge\ \text{v} \in \text{GMempCars)}
\end{aligned}$$

The first Select operation creates a set of all GM employees. The Image operation extracts the cars for each employee – resulting in a set of sets which is then Flattened. The final Select chooses, from the set of all vehicles, those that are manufactured by GM and owned by GM employees. We would like the final Selection to be simplified to :

$$\text{Answer} := \text{Select(GMempCars, }\lambda\text{v v.manu\_by.name = ``GM'')}$$

This simplification, represented in Figure 4.1b, could result from the knowledge that, in our model, all sets with member-type T are subsets of the extent of T. Since GMempCars has type Set[Vehicle], it must be a subset of Vehicles (the extent of type Vehicle). This simplification is similar to the

domain refinement technique in semantic query optimization [66]. In our problem, however, the simplification is based on knowledge about set inclusion and abstract data types, not necessarily a particular ADT. Subtyping relationships give similar information about set inclusion relationships. For example, in most object-oriented models, the set Managers is a subset of the set Employees. This results because Manager is a subtype of Employee, and the sets are both extents.

4.1.3 Subtyping and Static Type-checking

Subtyping information can also affect the applicability of transformations in a statically type-checked system. For example, consider the query Union(Students,Employees). In models without Union types, the application of this set operation, if it is even allowed, would normally result in a set having type Set[Person], where Person is the closest common supertype of types Student and Employee.[1] As a result, only properties of type Person are applicable, as far as static type-checking is concerned, to the result set. This modelling decision means that a query transformation distributing Union (or Intersection or Difference) over other operations is not applicable if static type-checking is required. In other words,

$$\text{Union}(\text{QueryOp}(S_1,p),\ \text{QueryOp}(S_2,p)) \neq \text{QueryOp}(\text{Union}(S_1,S_2),p)$$

since p may not be defined for the type of $\text{Union}(S_1,S_2)$.

For example, consider a query which asks for the managers of all Students and Employees. This query can be expressed as

$$\text{Union}(\text{Image}(\text{Students},\lambda s\ s.\text{mgr}),\text{Image}(\text{Employees},\lambda e\ e.\text{mgr}))$$

but not as

$$\text{Image}(\text{Union}(\text{Students},\text{Employees}),\lambda p\ p.\text{mgr})$$

because the union result has (static) type Set[Person], and the *mgr* property is not defined for type Person. The *mgr* property could be applied to the union result,though, because of late-binding of the methods to the objects. If the intersection (i.e., objects of type Student-Employee) is large, the latter expression might be the most efficient to process; applying *mgr* after the Union would avoid applying it twice to objects in the intersection. The type inference mechanism of Straube and Özsu addresses this problem by inferring the methods (such as *mgr*) that can apply to the result of a Union (or other operation) [144, 147].

4.1.4 Summary

These examples illustrate that knowledge about abstract data types and subtyping in the object-oriented model can affect query transformations. In order to support abstract data types an optimizer must be able to incorporate semantic query optimizations, such as those discussed here, while supporting standard query optimizations. An optimizer must also support the extensibility that is inherent in abstract data type construction. The addition of new types results in new operations and transformations that should be considered in the optimization process.

[1] Of course, the closest common supertype can be ambiguous in systems that allow a type to have multiple supertypes (i.e., similar to multiple inheritance). Also, if closest common supertypes are used, set operations in a system with multiple supertypes will not be associative because the result types of different associations cannot be guaranteed to be the same.

4.2 Complex Structures

The complex structure of objects means that languages to query the objects must have mechanisms for exploring their structure. In addition, languages that allow the creation of objects need mechanisms for building new structures. The exploration and creation of such structures can lead to the regular use of path expressions to navigate through a structure. Support for complex structures also results in languages that regularly use nested query expressions; i.e., variables from outer expressions are referenced in nested expressions.

4.2.1 Path Expressions

Path expressions imply an execution order for properties on the path. For example, the path *s.comp.name*, where *s* is in Students, implies the execution *name(comp(s))* – that is, apply the *comp* method to *s*, then apply *name* to the result.[2] This order may not, however, be the most efficient way to process the query. A path can sometimes be more efficiently processed using Join. For example, if there are very few companies it might be more efficient to first compute the *name* property for each company and store this result in a tuple. The path from a student to a company name would then involve joining Students with the company/name tuples by matching the *comp* property of a student with the company attribute of a tuple. An optimizer should be able to make (and evaluate) such transformations between path expressions and explicit joins. As we noted in Chapter 2, such transformations are addressed by [75], [79], [91], and [118], for example.

Another difficulty with path expressions is the definition of indices for such expressions. For example, suppose an index from departmental secretaries to employees is required; i.e., an index is required for the path *e.dept.mgr.secretary.name*.[3] One question that arises is how to store this index. For example, a single index from secretary name to employee could be maintained [22], or indices could be maintained for segments of the path and combined to give the whole path [79, 97, 111].

Another problem arises when trying to maintain such an index. All steps of a path must be considered when updating an index; i.e. changes to an index can be necessitated by changes to more information than just the ends of the path. For example, changes to a department or manager can affect the name-to-employee index, as well as changes to the employee or secretary. The presence of arbitrary methods in a path further complicates this problem, since the method results can be affected by changes to database information not apparent at the interface of the method. This is discussed more fully in the next section (4.3).

4.2.2 Common Subexpressions

In general we would like to optimize the use of common subexpressions in a query.[4] Many of these expressions will be path expressions. The use of common path expressions can complicate optimization because a common subexpression could be optimized differently over each instantiation of a common path. For example, consider the predicate

$$\text{v.owner.residence.city} = \text{`Boston'} \ \wedge \ \text{v.owner.employer.name} = \text{`GM'}$$

where v is a Vehicle.[5] The application of the *owner* method is common to both conjuncts of the predicate, and thus the expression *v.owner* is a common subexpression. However, if there is a

[2] These can be method applications, or just attribute references as in a complex object model.

[3] Assume there is no *mgr* property on Employees.

[4] The detection of such expressions is discussed in Section 4.4.

[5] We change the scheme slightly here to illustrate the problem.

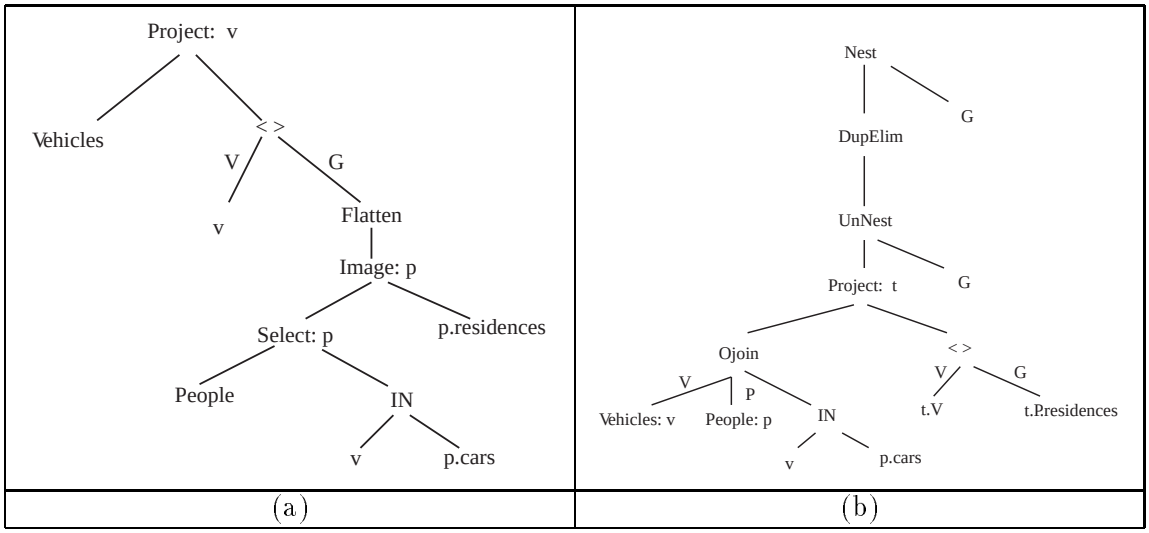

Figure 4.2: Query – Find all possible garaging locations for each vehicle.

path index on *v.owner.residence.city* we would not want to share the common path. Indeed, any transformation of v.owner on one path should not necessarily affect the other path. In general, any optimization of common subexpressions must determine whether the optimization transformation is applicable to a single path leading to the subexpression, or to all paths.

The formalism of Cluet and Delobel represents common subexpressions and optimizes them as single expressions [31]. To our knowledge, no research has been done to decide whether or not optimizations should be applied to every occurrence of an expression.

4.2.3 Nested Queries

A problem with nested query expressions is that query transformations may no longer be local transformations; they can involve query operators that are in different nesting levels. Consider for example a query to find the garaging locations for all Vehicles. The following algebraic expression satisfies the query by building a structure that stores a set of addresses with each vehicle.

Project(Vehicles,λv <(V: v),
(G: Flatten(Image(Select(People,λp v $\in$ p.cars),
λp p.residences))>)

This query is illustrated by the tree in Figure 4.2a. In this query the variable representing a Vehicle is referenced in the nested Select query. The $v \in p.cars$ predicate indicates that there is a relationship between Vehicles (represented by variable v) and People (represented by variable p) that could be effected with a Join operation.

The query expression illustrated in Figure 4.2b also satisfies the query. A difference in the solutions is that Project does a left outer join, so could result in vehicles related to empty sets of residences. In this example, the empty sets could easily be eliminated if necessary. There is a large body of research for dealing with outerjoin operations [38, 50, 120, 122].

The two query expressions differ significantly in their use of variables. In the first query, variable reference is nested, while in the second query all variables are local to the operation using them. The second query is similar to relational queries in which a Join of all relations aggregates the data,

then a Project operation builds the result relation. In this query, the data is basically collected using the Join operation, then the structure for the result is built with the Project, Nest and UnNest operations.

The transformation between the Project and Join versions of a query expression may require more than the application of context-free algebraic transformation rules. The arbitrary nesting of query expressions means that recognition of join relationships involves global knowledge about the expression. In addition, any transformations may have to preserve the structure of result objects. Again, producing such a structure requires knowledge which includes the effects of all parts of the query expression on that structure.

An optimizer should be able to generate, and work with, both the Project and Join query representations. The Join representation of the query might result, for example, from a mechanical translation from an SQL-like query. In this case it might be useful to translate to the Project version of the query to explore optimizations that cannot be recognized from the Join version. On the other hand, if the query is initially the Project expression it would be useful to generate the Join query, especially if there are efficient techniques for dealing with Join operations.

Processing of nested query expressions in SQL addresses situations where queries are nested in Where predicates [51, 38, 83, 109]. Proposals for handling nested queries in object-oriented SQL-like languages address nesting in From and Where clauses [31, 115]. The problem discussed here is equivalent to a query expression nested in an SQL Select clause. This problem is addressed by a new optimization strategy discussed in the example optimizer of Chapter 8 [148].

4.3 Methods and Encapsulation

The applicability of algebraic transformations to object-oriented queries is complicated by the object model. In general we have found that there are no universally applicable transformations, and that type and storage information about objects is necessary in order to decide the utility of an algebraic transformation. For example, in the object-oriented model, the cost of applying a Selection operation is an important factor when deciding whether to push the Select past a Join [42]. Although this is also a recognized problem with relational queries, it is not as prevalent and, in the relational model, pushing a Select operation past a Join operation is generally accepted to be a useful transformation. However, the presence of methods in a Selection predicate of an object-oriented query can mean that the cost of applying that operation depends on the cost of applying those methods.

Consider, for example, a query which matches managers with budgets of over one million dollars with students, studying at the same company, with grade point averages of more than 3.5. An SQL-like version of this query is:

```
Select   [RichManager: m, GoodStudent: s]
From     m in Managers
         s in Students
Where    m.budget ≥ 1000000 AND
         s.gpa ≥ 3.5 AND
         m.employer = s.comp
```

Applying relational heuristics to this query could give an algebraic expression such as:

```
Ojoin (Select(Managers, λm m.budget ≥ 1000000),
       Select(Students, λs s.gpa ≥ 3.5),
       RichManager, GoodStudent,
       λm,s m.employer = s.comp) )
```

This query joins Managers with budgets of over one million with Students who have appropriate grade point averages, then Selects over those pairs by matching the companies. Such a query expression results from strategies that push selections (on *budget* and *gpa* respectively) to the single relations to which they apply in an attempt to reduce the sizes of collections to be joined. The heuristic used is that joining is an expensive operation, but selections are straightforward or could even be done as part of the joining process.

However, consider the case where computing a manager's budget is an expensive operation. In this situation it could be more efficient to first join Students (perhaps selected by gpa, depending on the expense of that operation) with Managers by matching the companies. The join operation could reduce the number of managers to which the budget method must be applied.

This type of situation is certainly present in relational systems, but the presence of methods in the object-oriented model — with possibly arbitrary computations implementing the methods — means that strategies such as pushing Selection past Join will be less often applicable. One approach that avoids the cost of applying methods at query execution time is method precomputation [20, 78]. An alternative strategy for object-oriented models is to consider cost in the application of rewrite transformations. This would require cost models for high-level expressions, as noted in [59].

4.3.1 Method Cost

The determination of the cost of a query, or query operation, is complicated by the presence of methods and encapsulation. A query optimizer needs a way to ascertain method cost in the presence of encapsulation. If the optimizer is allowed to break encapsulation and look at the implementation of a method, then it must be able to understand that implementation. For example, a method could be written in the query language understood by the optimizer. The method code could then be merged with the query and managed by the query optimizer [15]. This approach, of course, limits the expressibility of methods to that of the query language.

If the optimizer gathers cost information about a method by querying the method itself [60] then type Method must define an interface that can provide the required information. As a simple example, type Method would have a property named *cost* which, when applied to a method instance, would return an expected cost for executing the method.

An alternative to determining cost of method applications, of course, is to store precomputed results in tuples [20, 78]. The costs of method application are then, in many cases, transferred to compile time.

Late-binding of methods also complicates the determination of method costs, since the implementation (and therefore the cost) of a method used in a query will not necessarily be known until query execution time. For example, suppose the gpa method for type Student directly accesses values in the representation of Student but the method is overridden in type Student-Employee by a more expensive method (perhaps computing an average using Student and Company information). The cost of applying the gpa method to members of Students can not be statically determined since some objects in the set could actually have type Student-Employee. The dynamic query evaluation plans of Graefe and Ward are designed to address this problem [62].

4.3.2 Indexing

The definition and maintenance of indexes is complicated greatly by the presence of methods. An index over even a single property implemented as a method could require the manipulation of arbitrarily many objects. This is similar to the problem with maintaining path indices discussed

in Section 4.2. A method could be implemented as a path, or some other arbitrary computation, over many database objects. Modifications to any of those objects can change the method result, and thus the index value.

For example, suppose Managers are indexed by their *budget* property, and suppose that property is implemented as a computation that includes retrieving the salaries of all employees in the department. Clearly, a change to the salary of any employee will affect the index on Managers. The problem here, then, is defining indexes so the scope of changes that can affect the index is known and, perhaps, preventing the definition of indexes that will require extensive maintenance. The former problem is similar to the problem of determining when precomputed methods are invalidated [78].

4.4 Object Identity

The identity of objects involves modelling and language decisions that can affect the optimization of queries. When objects have identities, there is a question as to what constitutes equality of two objects (see for example [82], [132]). This carries over to the language, where equality operations are used in predicates and where a decision must be made concerning the creation of new objects by a query. The creation of new objects can lead to new definitions for equivalence of queries that affect the transformations available to an optimizer. An optimizer for object-oriented models must be able to deal with the creation of new objects and with alternative definitions for equivalence.

4.4.1 Object Equality

Equality of objects in a query can refer to any definition of equality for type Object (e.g., identical, deep-equality) or to equality operations defined for a particular abstract data type. In a query language a variety of equality operations could be permitted in predicates. For example consider the operation $a.cars = b.cars$. Equality here might refer to identical sets (i.e. a and b reference exactly the same set of cars), to shallow equal sets (the sets of cars referenced by a and b contain exactly the same cars), or possibly even to sets with the same cardinality (if = is defined as such for type Set[Car]). In the presence of object identity, an equality operation is actually a method, and will have to be treated as such by an optimizer.

4.4.2 Object Creation

A major language decision that affects the optimization of queries is whether a query language can create new objects and, if so, whether those objects can have identities. If the language does not create objects, new relationships between objects cannot be built by queries. If the language only creates objects without identities, then it must be able to work with those objects as well as with objects with identity. On the other hand, if the language creates objects with identity then it must have mechanisms to determine new identities and manipulate those identities. For example, the EQUAL query language provides DupEliminate as an operation to manipulate unwanted duplication in objects that may be created by a query [133].

4.4.3 Query Equivalence

The creation of new objects with identity by a query can complicate the optimizer's task. For one thing, the optimizer may want a mechanism for deciding whether to create identities for objects in intermediate stages of a query. Also, the creation of objects with identity complicates the meaning

of equivalence of two queries. The result of a query can be a new collection of objects with a unique identity, and as a result even two responses to exactly the same query are not identical. The creation of new objects by a query language means that the structure of results as well as the data retrieved by a query must be considered when defining equivalence.

We have defined alternate notions for equivalence (see Chapter 3 and [132]) that must be recognized by an optimizer working with a language that builds objects with identity. For example, our weakest notion of equivalence states that two queries are equivalent if they respond with the same data, regardless of the logical structure of the objects returned. The query "For each student find all employees in the same department" could be answered by an Ojoin, returning Student/Employee pairs, or by a Project (followed by a Select to eliminate students matched with no employees) returning Student/Set[Employee] pairs.

The ability to define these alternatives might allow the optimizer to choose a more efficient method for solving a query. It also might give the optimizer more latitude when working with nested queries. For example, consider the query:

```
Select e
From   s in Students
       e in Employees
Where  e.mgr = s.mgr
```

which returns all employees who work for someone who also manages a student. A straightforward translation of this query to an algebra could give:

```
Project(Ojoin(Students,Employees,S,E,λs λe s.mgr = e.mgr),λt <(Emps,t.E)>)
```

Note that this translation assumes that the type to be returned can be Set[Tuple[Employee] rather than Set[Employee]. The assumption implies that a given employee can be represented many times in the result – once for each student with which it is associated in the Join. If an Image operation is chosen instead of the Project the result would not contain duplicates of individual employees.

An query optimizer that has the ability to make such transformations might find them useful in providing further opportunities for optimization. For example, an optimizer that can deal with weakly equivalent transformations can transform the previous query by substituting a Project operation for the Ojoin giving

```
Project(Select(Project(Students,λs <(S,s),(E,Select(Employees,λe s.mgr = e.mgr))>),
                λt NOT Empty(t.E)),
        λt <(E,t.E)>)
```

A further transformation could commute the outer Project and Select, since the Project retains the information needed for the Select:

```
Select(Project(Project(Students,λs <(S,s),(E,(Select(Employees,λe s.mgr = e.mgr)>),
                λt <(E,t.E)>,
        λt  NOT  Empty(t.E)))
```

The two Project operations can then be collapsed into one giving:

```
Select(Project(Students,λs <(E,Select(Employees,λe s.mgr = e.mgr))>)
       λt  NOT Empty(t.E))
```

The final result performs fewer operations than the original query, and also is concerned with fewer objects since there are no intermediate results. On the other hand, the final result has a different type than the initial query expression (i.e., Set[Tuple[Set[Employee]]] as opposed to Set[Tuple[Employee]]). The change in type information occurred when the Project was transformed to Ojoin using the weak equivalence. An optimizer could apply weaker equivalences to find better solutions, but must then be able to determine when such transformations are acceptable. This might involve the ability to restore the type of a result.

4.4.4 Common Subexpressions, Object Creation, and Equivalence

The creation of objects by a query complicates the determination of whether two query expressions can be considered to be the same – i.e., what are common subexpressions? If a query expression represents a constant (i.e., a named database object) or a variable, then it is straightforward to determine common subexpressions. For example, in the query *Ojoin(People,People,A,B,some predicate)* it is clear that both references to People refer to the same database set. However, if a query expression represents a path, or a nested query, it is not always clear when two syntactically equivalent expressions yield identical results.

Methods called in expressions in the language (e.g. in path expressions) may create new objects as their results. Depending on the database model, methods can written in arbitrary programming languages or may be queries. We assume that queries do not modify the database, but are providing information gathered from observations of the database. Thus, methods or queries are *observers*. However, we can identify two kinds of observers: 1) those that return only existing database objects and 2) those that generate new objects (e.g. queries). We will call functions that return existing database objects *pure observers*. Functions that generate new objects are *generative observers* (or, simply, *generators*).

In a path expression containing methods that are generative observers, we have the query equivalence problem. For example, in the query *Ojoin(Q, Q, etc.)* where Q is some nested subquery, the two executions of the Q subquery could create different objects and would therefore not be considered to be common subexpressions. In general we would assume that an optimizer would ignore different objects generated by multiple applications of a query expression. On the other hand, it is possible that an optimizer could make use of such occurrences.

4.5 Summary

In this chapter we have noted some of the optimization problems that accompany support for abstract data types, complex structures, encapsulation and methods, and object identity. These problems need to be considered by designers of database models and languages, since support for a feature will require solutions to the problems accompanying that feature. Solutions to some of the problems we have identified require optimization techniques that go beyond current relational optimization technology. An object-oriented query optimizer must be able to incorporate techniques for solving the different problems.

The variety of problems encountered when optimizing an object-oriented query, and the different approaches to solving these problems, are addressed by the architecture presented in the following chapters. We describe a new approach to optimizer extensibility that supports extensions to the collection of strategies for query transformation embodied in an optimizer, as well as the standard kinds of extensions supporting the data model. This approach is motivated by the identification of the problems discussed here and the different strategies that can be employed to solve some of these problems. We believe that the extensibility of the object-oriented database model will lead to

the identification of new problems in query optimization and, hopefully, new techniques for solving these problems. Our architecture for query optimization will allow the addition of new optimization strategies as they are developed.

Chapter 5

An Approach to Extensible Query Optimization

Optimization of a query Q is inherently a process of searching the space of all queries that are equivalent to Q. Typically, a given optimizer can only visit some portion of this space, since the set of transformation rules is usually incomplete, the cost of optimization must be bounded, and the optimizer control strategy limits the search. This situation is depicted in Figure 5.1.

As shown in the figure, an optimization strategy (i.e. the control strategy of an optimizer) determines, for any query Q, the equivalent queries that will be searched. The challenge here is to search a space that includes a good, or even best, equivalent query. Also, the optimization strategy must work for any query, since the same strategy is used regardless of the query being optimized.

The extensible nature of the object-oriented approach requires that the optimizer search space be expandable as a response to the new kinds of expressions that can be written. One approach to extending the search space is to "enlarge the flashlight beam", i.e. to expand the space searched by the optimization strategy. This is the approach taken by many extensible optimizers (e.g. [54], [59], [64], [140]). Generally, new rules for transforming query expressions are added to an optimizer and the same rule search strategy is used to find and apply these rules. The optimizer is extensible but the optimizer control strategy is fixed.

An alternate approach to optimizer extensibility is to allow the addition of new optimization strategies. This approach is depicted in Figure 5.2 and is the approach described in this chapter. In the next section we describe Epoq, a strategy extensible approach to query optimization. A simple example optimization using this approach is described in Section 5.2. In Section 5.3 we discuss the extensibility of an Epoq optimizer. We give a formal basis for the Epoq approach in Section 5.4, and summarize in Section 5.5.

5.1 Epoq : A Strategy Extensible Approach

The phrase *extensible optimizer* takes on a new meaning in the "multiple flashlight" approach of Figure 5.2. Extensibility now refers to the ability to enhance an optimizer with new strategies for manipulating query expressions. The collection of strategies in an optimizer can be extended in response to other extensions in the query system. This strategy extensibility is provided by an Epoq optimizer, and also motivates the approaches taken by the optimizers of [89] and [129].

The Epoq approach to strategy extensibility is provided with a modular architecture in which each module specifies a strategy for query optimization. An Epoq optimizer is a collection of modules, called *regions*, each of which can manipulate a query expression to discover some subset

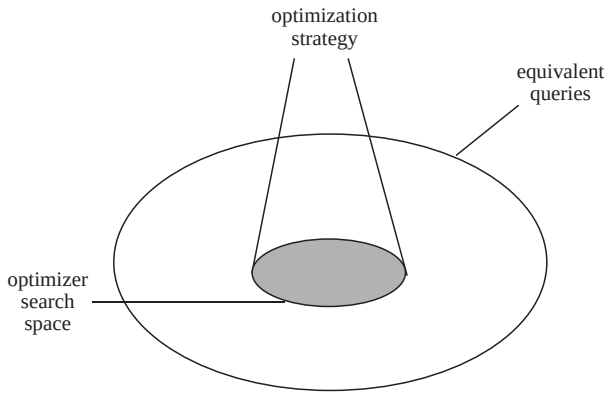

Figure 5.1: Query Optimization: A Search Problem

of the possible equivalent expressions. The regions can differ in the queries they can manipulate, in the manipulations they can perform, and in their goals for query manipulation.

Regions act as query transformations — they process a query expression to produce an equivalent query expression. The input and output expressions could be in the same language, or in different languages. For example, a region could do an algebraic rewrite, with the input and output queries expressed in the same algebra, or a region might do a translation from an algebra to access methods for the algebraic operators. A region may transform a query to improve the expected cost of a query, and in that case can be considered to be an optimizer. On the other hand, a region may transform a query without consideration of cost improvement. In Epoq, the term *query transformation* is used to mean any function that transforms a query to produce an equivalent query.

Each region defines a strategy for transforming query expressions. In an optimizer these strategies are combined to transform an input query expression to an expression with a better expected execution cost. The optimization of any given query is determined dynamically as some sequence of strategy applications. The strategies used, and the sequence of strategy applications, may be different for different query expressions (and are, indeed, expected to be different).

A major issue that must be addressed is the integration of these different strategies in an optimizer. In Epoq, integration is achieved through a hierarchical control relationship between regions. As illustrated in Figure 5.3, the regions in an optimizer can be nested to arbitrarily levels, forming a hierarchy. At the root of the hierarchy is a global optimization region that incorporates control for coordinating the regions below it. The global control might initially send the query to one or more regions; then decide to move the query from one region to another, until finally an acceptable "optimal" query form is produced. A region at any internal node of the hierarchy will coordinate other subordinate regions. Finally, at the bottom are primitive regions for which all transformations are internal.

For example, the three strategies indicated in Figure 5.2 could be represented by the three sibling regions at the first level of Figure 5.3. The root of the tree represents an optimizer incorporating these strategies, and one of the strategies is captured in a region that controls subordinate regions. The integration of subordinate regions is supported by a common interface between a parent and its children, and by a parent control that defines how the subordinate regions will be used in the optimization process. The interface is discussed in Section 5.1.2 and control over subordinate regions is discussed in Section 5.1.3.

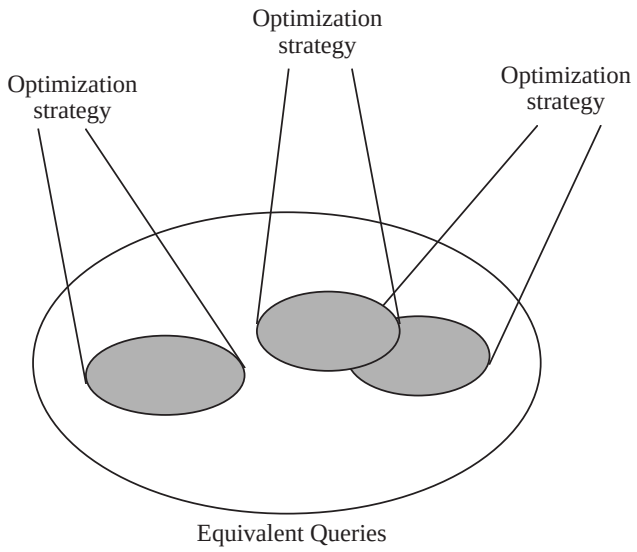

Figure 5.2: A new approach to optimizer extensibility.

The extensibility of an Epoq optimizer is founded on the ability to add new regions to the hierarchy. The addition of new regions corresponds to extending an optimizer with new strategies for query optimization. New regions are added as leaves of existing regions. The new regions satisfy the parent region interface and are integrated into the optimizer through the parent region control. In other words, the parent, as part of its own transformation strategy, uses the new region, as it does its other subordinate regions, to do transformations over queries. The extensibility of an Epoq optimizer is discussed further in Section 5.3.

5.1.1 Regions

A strategy for achieving some goal involving a query is encapsulated in a region. Some possible goals are to optimize a query expression, to find the most efficient way to process path expressions, to find an efficient ordering of join operations, to find an optimal ordering for conjuncts of a predicate, or to generate a query execution plan.

Different regions can provide different strategies for controlling the query transformation process in working toward the region's goal. With the goal of lowering expected query execution cost, for example, one region could be responsible for using algebraic rewrite rules to manipulate general query expressions while another region optimizes only Select queries with no nested query operations. One region might work on arbitrary query trees while another region might manipulate queries in a particular canonical form. The kinds of queries manipulated in a region, and the kinds of manipulations performed in a region, define the semantics of that region.

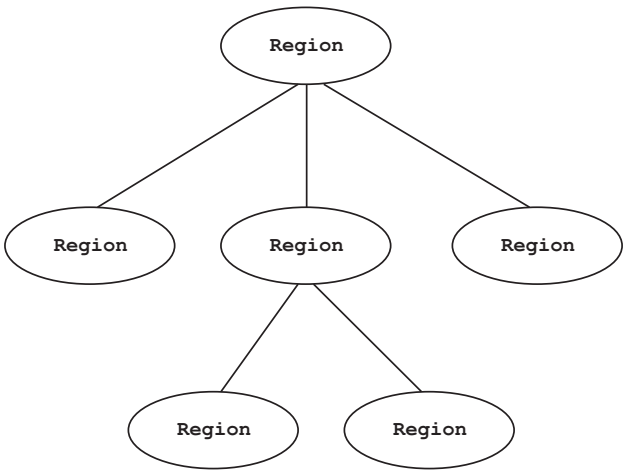

Figure 5.3: Integrating strategies

A region is characterized by

- a predicate describing the set of possible input queries that can be accepted by the region (i.e., a canonical query form for the region),
- a collection of transformations that can be performed on those queries,
- a means of control over the application of those transformations, and
- a goal predicate that characterizes the output queries produced by the region.

The characterizations of the input and output query forms are provided at the interface of a region. These characterizations supply information about the region that is useful to the parent in making decisions about executing subordinates. The characterization of input queries indicates whether a region is able to process a query, so can be used by a parent to eliminate from consideration regions that cannot process a given query. The characterization of output queries is useful to the parent control as it decides how to use its subordinates to achieve its own goals for query transformation.

The region's control strategy is defined by the transformations it can perform, and the control over the application of those transformations. Although this characterization of a region's implementation implies a control module that is separate from a collection of transformations (as in a rule-based optimizer, for example), that is not necessarily the case. We mean to say here, simply, that the implementation of a region involves the ability to transform query expressions. The transformations a region does internally are determined by the control strategy of the region considering the query it is processing.

Regions do not necessarily partition the space of query expressions — they may overlap in their sets of possible input queries, the transformations they can perform, or in the output queries they can produce. The possible redundancy in regions is managed by a parent region control as it decides about the execution of its subordinates.

The regions provide the possibility of segregating sets of transformations about which there is some common characteristic. For example, if bottom-up tree search is found to be better for Join queries, but top-down is better for manipulating query predicates, then each of those strategies

could be incorporated in a different region. Each region would indicate, through its input and output query characterizations, which kinds of queries it processes. A parent region uses these characterizations as it decides on region executions.

Regions also provide the ability to extend the optimizer with new strategies for optimization. A new strategy is represented by a new region that can be added to the optimizer. The addition of new regions is supported by a standardized region interface and by controls in parent regions that can use the interface information in making choices.

Region Architecture

The major components of a region are a control, an interface to parent regions, and an interface to child regions. The control embodies the region's strategy for achieving its goal. It includes components for decision-making, decision support, and query manipulation. Among other tasks, the decision-making part of the control must determine an order for applying query transformations, the decision-support manager provides information required for decision-making, and the query manager executes those transformations.

The parent interface interacts with a parent region to communicate information about the region and the query it processes. For example, it provides information to the parent about the region's goals and applicability to a particular query, obtains a query to process, and communicates results to the parent. The subordinate interface communicates with child regions to obtain information about those subordinates for the control and to process control requests for subordinate transformations.

This architecture for a region can be viewed as an abstraction in which a region has an interface and an implementation. The interface corresponds to the parent and child interfaces, and the implementation corresponds to the region control. The interface, therefore, describes communication between regions, and the implementation processes a query and provides the information communicated through the interface functions. In a very general sense, the implementation is an internal representation for the interface because it implements the functionalities that the interface claims to provide. For example, the interface will provide information about a query result and whether a goal has been achieved for that result — this information is computed by the control.

The Root and Interior Regions

Root and interior regions define a control that integrates other regions. This control is supported by a standard interface to child regions. Root and interior regions differ only in that an interior region's parent interface interacts with another region while the root region's parent interface interacts with the query processing system (usually the parser).

The interface of a region R to its children specifies the requirements any region must satisfy in order to be a child of R. These requirements allow R to communicate with its children. Similarly, the interface of R to its parent must satisfy that parent's requirements.

If a region's interface satisfies the requirements of more than one parent region it could, indeed, have more than one parent. The parent-child relationships between regions can form a directed acyclic graph, directed from parent to child.

The control of a region is responsible for transforming a query to meet the region's goal. This responsibility includes local manipulation of the query expression, as well as the delegation of responsibility for query transformation to its child regions.

In an interior region the execution of a transformation is actually the invocation of a child region to do some transformation. This is the major way in which a region differs from a rule-based optimizer, and the way in which an interior region differs from a leaf region. The actual invocation

of the child is made possible through the interface definition (section 5.1.2). Determining an order for executing child regions is the critical part of the control that makes the extensible optimizer work. This is discussed further in Section 5.1.3.

Leaf Regions

We expect that most of the actual manipulation of queries will be done by the leaf regions. Interior regions will, in general, delegate transformation tasks to subordinate regions. The leaf regions, obviously, do not delegate any transformations — the transformations performed by a leaf region when processing a query are all internal to the region. The control over these transformations may be designed following the architectural guidelines for an Epoq region, but this is not required. The only requirement of a leaf control (other than correctness, which is assumed) is that it provide the information needed at the interface.

Leaf regions are the way in which independently developed query processing strategies can be incorporated into an Epoq optimizer. A leaf region could, for example, be a module that processes path expressions, a module that manipulates nested queries, or even a rule-based optimizer. In an Epoq optimizer, each of these strategies can be effected by an arbitrary control that is completely enclosed within the leaf region.

Advantages of Regions

Separating the optimizer into regions provides a number of benefits in terms of optimizer efficiency, flexibility, and extensibility. A major advantage results from the fact that the region-based approach allows an optimizer to incorporate a number of different optimization strategies. These strategies can be used to customize the optimization process for a particular query — the execution order of strategies can react to the query being processed and to previous manipulations of the query.

Also, an optimizer can include specialized strategies that are specific to a small number of queries. This could also result in more efficient processing of a query, since strategies that are not useful for processing the query do not have to be executed.

The number of transformations that have to be considered by a region can be smaller, which could result in faster searches for transformations within a region. Also, the separation of transformations into regions allows for tuning of the individual regions, which itself could result in more efficient optimization.

The efficiency of optimization involves a performance trade-off between the modularity of the optimizer and the complexity (or simplicity) of individual modules. There is obviously cost involved in the interaction between regions. The performance requirements of maneuvering in the hierarchy need to be balanced with the performance of the individual regions.

The standard levels of optimization can be mixed in a region-based optimizer; i.e., there is not necessarily a fixed execution order for the processing steps of an optimizer. For example, if the appropriate regions and parent control are provided, access plan cost analyses can be mixed with semantic or algebraic manipulations of a query.

Similarly, different kinds of optimization can also be mixed in a region-based optimizer. In particular, we would expect some regions to do cost-based optimizations while other regions concentrate on heuristic-based optimizations.

A region-based optimizer provides the potential for experimenting with new strategies for optimization and with different ways to integrate these strategies. For example, it would be interesting to experiment with the parallel execution of regions.

Finally, the modularization of the transformation process into regions, each of which has a standard interface, makes extensibility of the optimization process possible. Such extensibility supports the extensibility of the object model by providing for the incorporation of new strategies for query optimization.

5.1.2 Interface

Three major aspects of an Epoq optimizer addressed by the interface are

1. communication between regions. The standard interface between a parent region and its children allows hierarchical communication between them.

2. optimizer extensibility. Any interior region (and thus the optimizer) can be extended by adding new child regions. These regions obey the standard interface defined for that parent.

3. control over transformation. The parent control needs to determine which child regions to call to perform query transformations. Information used in the parent's decision-making is provided to the control through the interface.

Extensibility and communication are supported primarily through the standardization of the interface. Parent control over transformations is supported by functions provided at the interface that supply information about a child to its parent.

A parent region places requirements that must be met by its subordinate regions. The parent and children pass query expressions back and forth as a query is transformed and, in general, the parent needs information to help guide its control of the transformation process. For example, a parent will need to know which goals a region can try to meet, the applicability of a region to a particular query, and whether a subordinate region successfully transformed a query. This information is provided to the parent by its children through the interface.

It is necessary that a parent and child place the same meaning on information that is passed between. For example, if a parent is looking for a "lower cost" transformation and a child has a goal of "lower cost" then they both need to mean the same thing. In the case where the semantics of the information is captured in a common metric, the matching of semantics can be done automatically. For example, a parent/child interface includes a single cost model (with its cost metric) and query representation, and goals expressed as computations on that model and representation could be automatically checked for consistency between the parent and child. Thus, a goal such as *cost(query-out)* $\leq$ *.90* $\times$ *cost(query-in)* (i.e., a goal of reducing query cost by at least 10 percent) would mean the same to a parent and child as long as they both used the same cost metric.

We can also envision situations in which a parent region can adjust its expectations of information it receives from particular children. For example, if a child region is asked to provide an estimate of execution time, then we of course assume that the region is using the same speed clock as the parent. If, in the parent's opinion, the region's estimates are consistently incorrect then the parent could, if desired, adjust future responses to consider the expected error. Similarly, if a child states a goal (e.g. "lower cost") we will assume the region uses the same meaning for the goal as the parent.[1]

In general, assuring consistency of semantics between a parent and child region is beyond the scope of this thesis and so will not be further discussed here. We assume that consistency of

[1] In the control design of the next chapter, the meaning of a goal is captured in its Success? function. This function is a method over a goal object, thus is available to all regions with access to the goal. As long as the regions use the function, consistency is maintained.

semantics is the responsibility of the system implementor. When a new region is installed as a child of an optimizer region, the implementor must insure that the information made available through the interface of that region matches the semantic requirements of the parent.

This is not an unreasonable expectation, since one would expect an optimizer implementor to have a thorough understanding of the intended purpose of different regions. The addition of regions to an optimizer requires that the implementor know where to place those regions in the hierarchy. The placement of regions also requires the implementor to anticipate the expected use of new regions (i.e., strategies). The standardization of the interface between parent and child regions provides the mechanism for interaction between regions and for extensibility; the standardization of semantics must be provided by the optimizer implementor.

The exact functionalities available for interaction between a parent and child are determined, in large part, by the control of the parent region. If the control uses the goals declared by subordinates to help in choosing region execution orders, then (obviously) goals need to be available at the interface. If the control uses estimates of region execution time to choose among regions, then those estimates need to be provided by the subordinates. If a control wants to supervise the overall execution of the optimizer, then it may pass execution limits to a subordinate through the interface.

The minimum functionality of interaction is that of passing a query expression back and forth. A parent region hands a query to a child for processing, and receives back an equivalent query. This functionality is supported by a common representation for query expressions.

Query Representation

The control strategy of a region, and the queries that can be represented, depend in part on the internal representation used for queries. The interaction between any pair of regions includes the transfer of a commonly recognized query representation. Since a query is initially processed in the root optimizer region, it is required that queries expressible in any of the query languages used in the optimizer be representable in the root region representation. Also, extensibility of the model, algebra, and optimizer requires that the query representation be extensible. We discuss briefly here, and present in more detail in Chapter 7, a query representation that meets these requirements.

In an Epoq optimizer a query expression is represented as a tree[2] of function and data nodes. Data nodes represent the data that is manipulated as part of executing a query, and function nodes represent actions on data. Data nodes represent objects — either objects already in the database, objects built by a query or subquery, or objects built by applying methods to other objects. Every function node will have some number of input data nodes, representing the parameters to the function, and one output data node representing the function output. Edges of a tree connect a function node with its input and output data. In ENCORE, as in most object-oriented models, all operations in the query algebra, and all operations over database objects, can be represented as functions.

Figure 5.4 on page 70 (and other figures in Section 5.2) illustrates an Epoq representation, although, to simplify the picture, internal data nodes are not shown. Only function nodes and leaf data nodes are represented in Figure 5.4. The leaf data nodes represent database objects, while interior data nodes represent function results. As can be seen, the representation (without interior data nodes) is similar to a syntax tree for the query expression. This supports extensions to the query model and algebra, and also allows the representation of nested query predicates as explicit subtrees.

Communication between parent and child regions is also supported by annotations to the nodes and edges of a query tree (see Figure 5.5). The annotations can be used by the regions. For

[2] Actually we will see in Section 7.6 that a query expressions is represented as a directed acyclic graph.

example, a data node can be annotated with type information about the data. A node representing a set object could be annotated with information about storage structures or access paths to the object. It might also be useful to know whether there are indices on the set, or if the set is ordered in any interesting way. A node representing a method could be annotated with the estimated cost of applying the method. Since query operations are also methods, cost estimates for those nodes might be useful in assessing the utility of transformations.

Although we would expect all regions in an optimizer to support the same representation, it is certainly possible for a given parent and all of its children to support a different query representation than that supported by some other subtree of regions. Also, if desired, a region could use a different representation internally (and perhaps for passing a query to its children) than the representation it receives from its parent. In this case the region would be responsible for any translation between representations.

Cost Model

The ultimate goal of any optimizer is to reduce the expected cost of executing a query. This is the goal of the root region in an Epoq optimizer, and will also be a goal of many subordinate regions. Expected query execution cost is evaluated using some cost model over query expressions. Since there are not, to date, any appropriate cost models for object-oriented systems like ENCORE, Epoq provides the mechanism for incorporating a cost model into an optimizer.

In an Epoq optimizer, information about expected cost of a query is carried along with the query as annotations to the representation. Annotations provide the flexibility for an optimizer implementor to choose (or customize) a cost model, and also provide extensibility to support cost models that may change as the data model and query language are extended. Part of installing a cost model, then, is to make the appropriate annotations available and to provide methods to access the new annotations appropriately.

A result of representing cost using annotations is that cost information is available for use by all regions. Such information is traditionally used to translate a query algebra expression into an access plan, but could also be useful in algebraic rewrite. For example, a strategy that manipulates path expressions could use information about expected cost of user methods in the path to help determine efficient rewrites of the path expression.

A requirement of a cost model over a query representation is that it cover any of the situations that can be represented. For example, our representation can be used to depict queries expressed in a high-level algebra as well as access plans (with low-level operations). In order to be useful, a cost model would have to handle both of these situations.

One consequence of representing cost information in annotations is that the annotations will often need to be updated during the optimization process; as the expression of a query changes, the cost associated with that expression may change. We will, for now, assume that all information needed to assess query cost is independent of the region processing the query. Thus, updating query cost can be done as a method over the representation. This method can be called by any region that needs cost information for the query it is processing. By modelling it this way we can either explicitly call the method from a region, or allow active processes to work behind-the-scenes to keep the query representation up-to-date in terms of cost. In this way, regions that choose not to work with the cost annotations can modify queries without regard to the cost effect that will be visible to other regions.

It is possible for a region to use a different cost model than that of its parent, although in most cases we would expect all regions to use the same cost model. If a region does choose to use its own cost model, however, the region is responsible for restoring the query representation and cost

annotations to the form and model required by the parent. Thus a region that uses a different cost model than its parent is responsible for any translations between cost models.

5.1.3 Region Control

The strategy for achieving some goal is encapsulated inside a region. This strategy is implemented through the region's control over the application of query transformations. In an Epoq optimizer, subordinate regions act as query transformations, and a parent region controls the application of those transformations.

Within a region, some of the control tasks are to:

- store the query expression (and alternatives)
- communicate with parent and child modules
- apply transformations to a query or subquery
- determine whether a goal has been achieved
- decide when query processing is complete

The first two tasks are bookkeeping tasks for the region, and were discussed in Sections 5.1.1 and 5.1.2. The other tasks relate directly to a region's use of its subordinate regions to transform query expressions. These tasks are discussed in this section.

The main goal of defining a control is for the optimizer itself to run more efficiently. A region could apply transformations exhaustively, but that would require excessive time, as well as excessive space for keeping track of results. Thus, a region's decisions about the application of child regions (i.e. transformations) are crucial to the efficient operation of an optimizer.

Control decisions

The major decisions that need to be made by a region are 1) which query or subquery to process and 2) which subordinate region to execute next. A region receives a single query to transform, as well as a goal for the transformation, and needs to decide what transformations to apply to the query, or any subqueries, in order to achieve its goal.[3] Throughout the transformation process a region will usually maintain a number of alternative query expressions, and will work on different of those expressions at different points in its processing.

One way to approach the decision process is to pair query expressions with applicable transformations (i.e., applicable regions), then select an expression/region pair to execute. Such a control is like a search through transformation rules, where the rules are the child regions. One difficulty in doing this is in determining when a region (rule) is the right one to apply to a query. Normally, rule-based optimizers do pattern matching (and usually condition testing) of the query expression with the left hand sides of rules, then perform some conflict resolution if more than one rule matches a query (e.g. assign weights to rules as in [54]). This approach is not completely satisfactory for an Epoq optimizer because the regions do not behave as precisely as rules behave.

[3]The dag representation for query expressions allows more than one query alternative to be represented at the same time. As a result, it is possible that a region could send a group of equivalent queries to a subordinate for processing. This brings up some interesting possibilities for region actions. For example, there might be a specialized region that can choose between alternative query expressions, or one that can merge equivalent expressions to give an efficient single expression. An exploration of these possibilities will not be undertaken here though, and we will assume that every region receives a single query to transform.

In a rule-based optimizer, rules provide complete information to a search engine, and can be applied by a rule execution process. In an Epoq optimizer, a region, behaving as a rule, provides incomplete information to its parent, and applies itself. The result of a region execution is returned to a parent, but the actual execution of the region is done independently of the parent processing. As a result, a region may not achieve its goal and may return a message to its parent indicating such a failure.

An alternative to rule search is to view region executions as actions in an optimization planning system. The decisions that need to be made (which region to execute, which query expression to manipulate) are managed by a planning system that is driven by its own planning rules. The planning rules are heuristics about orderings of region applications that will (hopefully) achieve the region's goals. Thus, the planning system is planning the execution of the optimizer. Since, in Epoq, a region may fail to transform a query (i.e. fail to achieve the region's goal), the planning process cannot proceed independently of the region results and is thus interleaved with region execution. This is discussed in more detail in Chapter 6.

Interface requirements

The decision-making component of a region's control is supported by information available at the interface of subordinate regions. Indeed, a region's control over its subordinates places requirements on that interface that must be met by subordinate regions.

For example, if a control wants to avoid executing regions that will not be able to transform a particular query then it can require each subordinate to provide information about the queries to which it can be applied. A region's applicability information should not reject query expressions that it can transform, but may accept expressions that it later discovers it cannot successfully transform. Such applicability information is required of all Epoq regions.

A region must also be able to provide information about its goals. Before executing a subordinate region, a control can use information about expected goals of a region to help decide which region to execute. After executing a subordinate, a control will need to know whether the region has attained the expected goal. In other words, the parent control will need to know whether the region has transformed the query as anticipated.

The specifics of the control of a region may place additional requirements on the information a subordinate must provide. For example, if a region's control is trying to transform a query quickly, then it may require that subordinates provide information about their expected execution time. This information would help the control decide which subordinate to execute to transform the query.

Goals and Success

The *goals* of a region characterize the output queries that can be produced by a region. Given a particular query, *success* indicates whether the region was able to attain a particular goal for that query. For example, if a region's goal is to lower cost, and the result query computed by the region has a lower cost than the input query, then the region was successful at achieving its goal.

Goals may describe an absolute query state, or a relative state. For example, 'lower cost' is a goal describing a relative query state — the result is compared to the input query. 'Join query' is a goal describing an absolute query state — a result is a join query if it contains join operations. Goals describing absolute query states may actually be met before a region does any transformation of a query. For example, if a query already contains join operations, the 'join query' goal is already met.

A region may not be successful at achieving a goal for a particular query. If a region thinks it is applicable to a query but finds, once it starts processing, that it cannot transform a query as expected then it may not achieve its goal on that particular query. For example, if a region whose goal is to improve expected cost of a query gets, as input, an optimal query expression, then it will not be able to attain its goal.

A parent region may choose, from the goals a subordinate region can try to attain, a goal to be achieved on a given query. In general, the parent may not define new goals for subordinates, since the subordinate region may not have the control knowledge to work towards those goals. A definition of goals that supports a parent's desire to modify a subordinate's goals might be interesting future work. At this point, however, any modification of goals must be expressed as termination conditions (see next subsection).

The parent's goal choice must be passed, along with the query, to a subordinate through the interface. The subordinate region must return an indication of success or failure to achieve the goal when the query result is returned. A parent region control needs to know whether a subordinate was successful in attaining the expected goal so it can decide how to further proceed with query processing.

Termination

Termination refers to stopping the execution of a region. In general a region will have its own internal criteria for termination, since termination is an integral part the region's control. Termination may be related to success; for example, a region may terminate processing a query when it discovers it is successful at achieving its goal. However, termination will often be conditional on more than success; for example, a region with a goal to lower cost will usually not quit as soon as the cost is lower, but will continue to try to improve the cost until it decides that further work will not be cost effective. In all cases, termination must involve conditions that are independent of success. A region will not necessarily achieve success, so termination conditions must ensure that the region stops regardless of success at achieving a goal.

A parent region may want to have some control over the execution of a subordinate and thus may want to be able to indicate conditions for termination. A parent region, for example, may restrict a subordinate to execute for a certain amount of time, to apply only a certain number of transformations to a query, or to produce some limited number of alternative equivalent queries. In order to allow this type of parental control, a region's condition for termination must be a disjunction of the conditions defined by the region and the conditions added by the parent at execution time. In effect, the parent's requirements for a subordinate's termination can override the subordinate's conditions when necessary.

Although termination conditions are expected to reflect the execution of a region, and will usually be independent of the region's goals and success, termination conditions can be used by a parent as a limited way to specify goals the parent wants a region to meet. As noted in the previous subsection, a parent cannot arbitrarily set goals for a subordinate since the child region cannot necessarily modify its control sufficiently to address the goals. However, a parent can specify a termination condition for the region that the parent, not the child, treats as a goal. The child region would indicate success relative to its own goals, and the parent could check the termination condition that it wants to treat as a goal to see if that condition/goal is met. For example, a parent may be satisfied with a twenty percent cost improvement. It could specify such an improvement as a termination condition, possibly causing the region to terminate before the child recognizes success. The parent would then have to check the result to see if that condition is met.

5.1.4 From abstract query to access plan

The optimization of a query takes an expression in a high-level query language and finds a good plan, in a lower-level language, for accessing the database to execute the query and produce the requested result. This process usually involves a sequence of stages, each of which manipulates a query expression in some way. Some of these stages may have different input and output query languages. For example, the methodology of Straube [144] first manipulates a calculus query, later transforms it to an algebraic query, then, after manipulating the algebraic query in different ways, transforms that to an access plan in a language for object access. Almost all query optimizers (extensible or not) process a query through some fixed sequence of optimization stages (e.g., [45], [54], [56], [64], [129], [130], [142], [152]).

An Epoq optimizer can simulate this kind of sequential processing in a straightforward way. For example, each stage of processing could be represented by a separate region, with a single level of control over those regions responsible for executing them in the desired order. The higher-level control region would make no decisions about which region to execute next — the execution order is a fixed sequence. A region in this sequence might have its own subordinate regions. For example, in the Straube approach [144] an algebraic expression is processed in two stages; the first checks type consistency in the expression and the second applies equivalence preserving rewrite rules to the expression. In Epoq these could both be subordinates of a region that manipulates algebraic query expressions.

Of course, the simulation of sequential processing of queries with Epoq does not use the potential of the Epoq approach. Although an Epoq optimizer would be expected to have regions that translate from one language to another, as well as regions that work within a single language, the flexibility of the region-based approach allows for regions that may understand different languages. Also, regions that understand the same language may be combined in different ways for different queries, and regions that translate from one language to another do not necessarily have to be executed sequentially. Although there is an implied sequence between groups of regions (i.e., regions that only understand queries in language A could not immediately follow any region that only produces queries in some different language B), within groups of regions the order of execution can respond to the query being manipulated. For example, regions which consume and produce queries in only language A (i.e., rewrite regions for A) can be executed in different orders for different queries depending on their goals, the query, and the strategy of their parent region.

In addition, Epoq allows for regions that can understand more than one language and whose execution can be intermingled with the executions of regions understanding different languages. This could be useful, for example, in developing strategies that can use access plan information to help guide rewrite. Such strategies could alternate between higher-level algebraic rewrite and lower-level access plan generation, and perhaps use this ability to generate more efficient plans (as suggested in [144], for example).

5.1.5 Comparison with other optimizers

The main difference between the Epoq approach to optimization and other approaches for extensible or object-oriented optimization systems is that Epoq provides for extensibility of the optimization process itself. As noted earlier, most extensible optimizers (e.g. [45], [54], [56], [64], [140]) provide a fixed strategy for searching for and applying rules for query transformation. In other words, although the possible optimizer results can be extended, the optimization process itself is fixed. This is analogous to the single flashlight picture in the introduction to this chapter. Proposals for object-oriented optimizers either use one of these extensible approaches [15] or provide some fixed

sequence of optimizer processing strategies [29, 81, 146].

The Epoq approach is motivated by the desire to extend an optimizer with new strategies for optimization. In other words, the optimization process itself is extended. Optimizer strategy extensibility also motivates the approaches of Lanzelotte and Valduriez [89] and Sciore and Sieg [129], so we will discuss these in more detail here.

Sciore and Sieg

Sciore and Sieg propose a modularization of the optimization process and describe an optimizer generator to build the modules [129, 137]. An optimizer is generated as a term-rewriting system in which rules are grouped into modules. In their approach the rules, and the strategy for searching for and applying rules, are separate. As a result, different modules can have different control over rule search. This achieves the goal of providing different strategies for different processing within an optimizer, but all strategies must be implementable as rule systems.

Although Sciore and Sieg support extending an optimizer with new strategies, the way in which these strategies are integrated into the optimizer is very different from the Epoq approach. In the Sciore and Sieg approach, modules interact with each other in ways that are fixed when the modules, and the rules, are written. A module declares a predecessor and a successor module, and rewrite rules may contain calls to other modules. Thus, a query moves through an "assembly line" of modules that is fixed regardless of the characteristics of the query being processed. In contrast, in Epoq control over the execution order of regions is separated from the regions being controlled. This modularization results in a control that can respond to the particular query being processed and to the dynamics of the processing of that query.

Lanzelotte and Valduriez

Lanzelotte and Valduriez address the problem of customizing the optimization process to a particular query by focussing on an extensible way to define strategies for manipulating query expressions [89]. These strategies are modelled as search strategies over a space of equivalent query trees, and each search strategy is implemented as manipulations over such a representation. Different search strategies are related through a sub-type hierarchy of strategies, with higher level specifications describing the methods present in a search strategy and lower level specializations (i.e. the specific strategies) implementing these methods (in different ways). A particular strategy can be modified by changing the implementation of any of its methods.

Different strategies are integrated in the sense that they all specialize a common model for search strategies. The common model is the search strategy for the optimizer and, at optimizer execution time, a specialization of the strategy can be used to process a particular query. The search strategy specializations are analogous to our leaf regions (and to the modules of Sciore and Sieg) and, indeed, may provide useful tools for specifying the implementation of regions. However, this work does not address the integration of the different strategies to process a single query at optimizer execution time. Given a query to process, one of the strategies present in an optimizer is chosen to optimize that query.

5.2 An example

To illustrate the Epoq approach we will follow an example query through the optimization process. For the purposes of this example, assume that the optimizer has the following regions:

1. Cost-guided — The goal of this region is to find a query expression with lower expected execution cost — i.e. this region is an optimizer. Syntactic transformation rules are used to manipulate arbitrary query trees. Application of the rules is directed by cost information about the operations and data involved in the query.

2. Join conversion — In this region an arbitrary query can be transformed into a query in a canonical join form; i.e. a form in which join operations are grouped at the bottom of the query tree. Such a transformation, of course, may not be possible — in which case the input query is returned as the result. The goal of the region is to find the canonical form, if possible, disregarding any cost aspects of the query. The expectation is that this form will lead to further, cost-effective transformation by other regions.

3. Join reordering — This region manipulates queries that are expressed in terms of Join operations to determine good join orderings and access methods. The goal is to find a lower-cost query.

We will assume that these are the only regions in our optimizer, although a full optimizer would be expected to have regions handling a large variety of problems encountered in object queries (e.g. path expressions or semantic optimizations).[4]

The regions in this example each represent a different strategy for query manipulation. The cost-guided region works similarly to existing rule-based optimizers that choose and apply transformations that are expected to reduce the cost of query execution. Rules about join reordering are not included here, since such transformations are probably better addressed by the join reordering region which uses a dynamic programming algorithm to manipulate join orderings. The join conversion region simply performs a query transformation. It is not concerned with improvement in the cost of query execution and may not even be able to discover any join relationships in a query.

The query we will manipulate is based on the scheme of Table 4.1[5] and is stated as follows:

> For each vehicle, list all possible drivers. Anyone who is over the age of 15 and lives in the same house as an owner of the car is a possible driver.[6]

This query can be expressed algebraically as

Project(Vehicles,λv <V:v,
Drivers: Flatten(Image(Select(People,λp v $\in$ p.cars),
λo Select(People,λp_1 p_1.age > 15
$\wedge$ p_1.residence = o.residence)))>)

Each Vehicle is matched with a set of people who are considered to be drivers of the vehicle. The result has type **Set[Tuple[V:Vehicle,Drivers:Set[Person]]]**. To simplify the explanation of the problem, assume that each vehicle has at least one owner. This means that the results will never involve empty sets of Drivers. This simplification means that we do not have to deal here with the outerjoin semantics of Project. Solutions to problems dealing with outerjoins can be found elsewhere (e.g.[38]).

Assume that this query enters the optimizer as the tree in Figure 5.4. The optimizer first annotates the tree with cost information such as ordering, indices and other storage access structures,

[4]This example is intended to illustrate how an optimizer could work on a query. A more complete optimizer, with more levels of regions and controls specified for most levels, is included in Chapter 8.

[5]The **residence** property of type Person is modified for this example to return a single residence.

[6]We call this the "insurance query" since, in Massachusetts, drivers who live with a car's owner may not be covered by the owner's insurance policy unless they are listed on that policy.

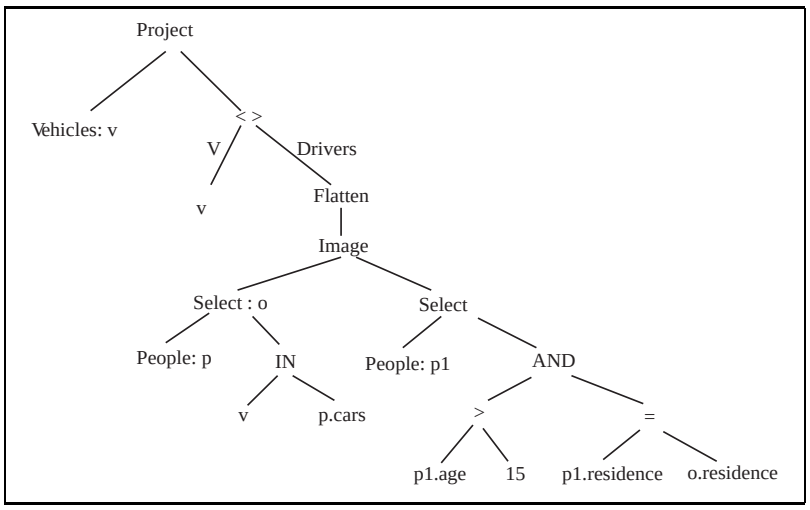

Figure 5.4: Query – List all possible drivers of each vehicle.

sizes of set objects (input and output), execution cost of methods, and selectivity of predicates (Figure 5.5).[7] For this example assume there are one million vehicles, two million people, and one half million residences. This information is reflected in the **size** annotations and in the **sel**ectivity of the residence comparison (there are four people per residence on average). We will also assume that half of the people are over the age of 15 (**sel**ectivity of $p_1.age > 15 = .5$), that half of the cars are jointly owned by two people and half of the cars are singly-owned (reflected in the average **size** of the output of p.cars). Assume that the *residence*, *cars* and *age* methods of the **Person** type all have low execution costs. This is indicated in the cost annotations for those methods. The People set is **order**ed by SSN (a user id number) and the Vehicles set is ordered by OID (an internal object identifier). The Vehicles set can be accessed through the People set (i.e. p.cars), as indicated in the **access** annotation.

The annotated query is sent by the global controller to the cost-guided transformation region where the cost information can be used to assist in choosing the transformations to apply to the query expression. This region is using a control strategy similar to hill-climbing where transformations are chosen that are expected to improve the expected execution cost of the query expression. The following transformation is applied to the query to yield the result in Figure 5.6[8]:

$$Select(S, \lambda s\ p_1(s) \wedge p_2(s)) \equiv Select(Select(S, \lambda s\ p_1(s)), \lambda s\ p_2(s))$$

The transformation essentially chooses an ordering for the Selection predicates. In this case, the $age > 15$ predicate is chosen to be executed first because the cost of applying the *age* method is low and the predicate is independent of the bound variable for the Image operation. The latter condition means that the query

$$\text{Select(People, } \lambda \text{p}_2\ \text{p}_2\text{.age} > 15)$$

can be pre-processed and accessed as a constant in the Select function nested in the Image query.

[7] This example sketches the annotated query representation. The representation is discussed in more detail in Chapter 7.

[8] To simplify the figures we will not annotate the nodes, and will discuss any relevant changes in annotations in the text.

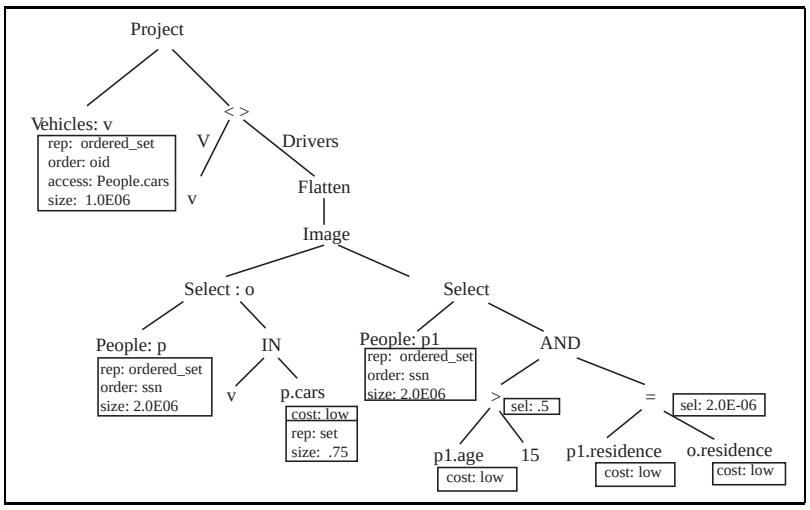

Figure 5.5: Annotated query.

For different query annotations, a different transformation or predicate ordering might have been chosen within the region. For example, if the age method were costly, if there was an index on a person's residence, or if residences were clustered, it might have been better to evaluate the residence-checking predicate first. Such an evaluation would do an indexed retrieval of a small number of co-residents, each of which could then be checked for the age requirement.

At this point the region control finds no other useful algebraic transformations to apply to the query, so the updated tree is returned to the global optimizer control. The global control then sends the query to the join conversion region. This region's control searches the tree for predicates that would indicate joins and finds two: $p_1.residence = o.residence$ and $v \in p.cars$. These predicates each match two sets in the query (the People set is matched with the subset of People over 15 and the Vehicles set is matched with the People set) indicating that the query could be expressed using Join operations. The query can thus be converted to a representation in a canonical join form as shown in Figure 5.7. In this representation, all join operations are clustered at the base of the tree, and operations that manipulate the structure of the query result (Project, DupElim and Nest in this example) are pushed to the top of the tree.

Although this region transformed the query, the transformation that was applied was not a context-free algebraic rewrite as in the cost-guided region, but involved a tree transformation procedure written in a general programming language. The control of this region simply checks to see if the transformation is appropriate (by checking for join predicates) then executes the transformation procedure. This region does not consider expected execution cost of the result and, indeed, may produce a query expression that is more expensive to execute than the input expression. On the other hand, the output from this region may offer more opportunities for optimization in other regions.

The latter is the case in this example. The canonical join query is sent to the join reordering region where the order of the join operations is modified and join methods are recommended. In this region, a dynamic programming strategy is used to search for good join orderings (similarly to the System R strategy [130]). The region eventually settles on the join ordering represented in the query of Figure 5.8 (note that annotations, including join method choices, are not shown in this figure). The reordering here is chosen on the basis of the direct access from the People to

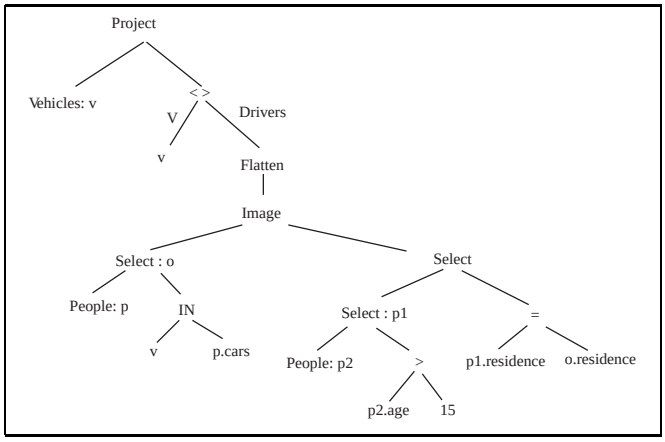

Figure 5.6: Transformation of Insurance Query

the Vehicles set and the expected cardinalities of the result sets. Each member of the People set is accessed once to retrieve the *cars* property. An average of .75 cars per person will be directly retrieved (recall that half of the people are too young to own cars, and half of the cars are jointly owned), resulting in 1.5 million pairs to be matched by residence to the set of People over 15. The Join methods chosen take advantage of the direct access between sets. Alternate join sequences are not optimal since they would match the two People sets with no indexes, resulting in four million pairs of people to access into the Vehicles set.

The reordered join query is returned to the optimizer control, which can then decide to send it to another region. In this example, suppose the optimizer chooses to halt and select the query tree which presents the most cost effective solution. In this case, the result of the join reordering region would be found to be best and this solution would be returned to the query processing system.

This example illustrates a simple use of the multiple region architecture for query optimization. The regions used in this example differ in the kinds of transformations they can apply and in the control used to apply those transformations. The join conversion region has a simple sequential control and can apply a single transformation procedure to a query. The cost-guided region is modelled after rule-based query optimizers extended to consider cost information in the transformation application process; the transformation rules are based on the algebraic properties of the operations involved in the query. Cost data about those operations provides further information used by the hill-climbing strategy of the region to control application of the transformation rules. We would not expect this region to consider join ordering rules, since join reordering is better handled by the dynamic programming strategy of the join reordering region. Such a strategy gives the same result as applying rules, but can be more efficient since it is tuned to the join reordering transformation process.

In this example the optimizer consisted of three leaf regions integrated by a parent region. The parent region control (global control), in this example, sent the query sequentially through the child regions. The cost-guided and join conversion regions were applicable to the initial query, and the control chose the cost-guided region as the first to apply. Rather than apply it twice in a row, the control chose the join conversion region to apply next. After join conversion, either cost-guided or join reordering could have been done next — the control chose to do join reordering. Rather than continue with another iteration of the cost-guided region, the control chose to terminate after executing the three regions and chose, from all results, the result which appeared to be the most

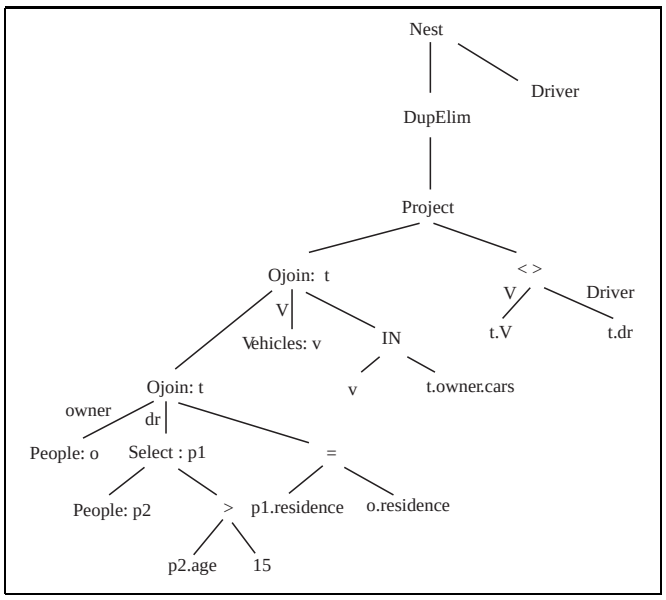

Figure 5.7: Join Conversion of Insurance Query

cost effective method for processing the query. This region execution ordering depended on both the characteristics of the query, and the parent region's internal control. A different input query, or a different parent control strategy, could have resulted in a different execution ordering of the three regions in this optimizer.

5.3 Extensibility

Extensibility in query optimization refers to the ability of an optimizer to respond to changes in the query system. Some of the components to which an optimizer needs to respond include:

- the data model – a data model is extended through the addition of new types. These types are the base upon which the database schema can be defined.
- the query algebra – new types in a model will result in new operations over objects of those types.
- algebraic transformation rules – new rules support new algebraic operations.
- data access methods – new access methods support new storage structures or added algebraic operations.
- cost model – new, or expanded, cost information may be needed to support additions to types, algebra and access methods.

Traditionally, such changes are supported by modifications to the information about the query system that is provided to an optimizer.

An Epoq optimizer supports changes such as these in the traditional ways, but it also offers a new response to changes in the query system. In Epoq, the collection of strategies used for optimizing queries can be extended in response to other extensions in the query system.

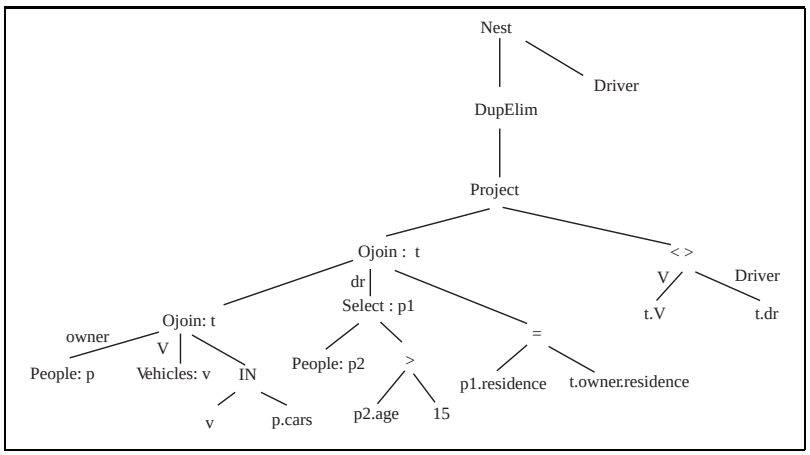

Figure 5.8: Query result after join reordering.

In Epoq, changes in the data model and algebra will be reflected in the query representation and, possibly, in other areas of the region interface. Such changes can also result in the addition of new regions. Changes to the model and algebra will involve new operations which must be represented internally to the optimizer. Since the operations are part of the query language, they will be represented by function nodes in the query representation. The operations must also be represented in the cost model.

The additions to the language may result in queries that can be manipulated by existing regions. In this case those regions must describe, in their characterization of applicability, that they can manipulate the new expressions. This could require modifications to the applicability information provided by a region to its parent.

Modifications to the algebra could also result in changes to existing regions. For example, a region with transformations expressed as rules, as in existing rule-based optimizers, might need to be modified by adding new transformation rules for the new operations.

The addition of new access methods might result from work in improving access to databases under an existing model, or might result from the addition of new data types and operations. In either case, new access methods could be treated in the same way as new algebraic operations. They may be handled by existing regions, or may result in the addition of new regions that employ new strategies for determining when to use the access methods.

Most modifications to the language will result in modifications to the cost model. These modifications can require changes to the set of available annotations of the query representation to reflect new kinds of cost information.

Any modification to the query system could result in new strategies for optimizing query expressions. These new strategies will normally be implemented as new regions in an optimizer. For example, if the ability to express paths was added to a language, we might want to add regions to manipulate path expressions.

The addition of a new region requires that the system implementor know where the strategy might fit in relation to other strategies; i.e., an implementor must choose a parent (or parents) for the new region. In many cases, adding a new region will involve only the requirement that the interface of the region be implemented to match the requirements of the parent. However, if a new strategy (and therefore new region) involves the definition of new goals for query transformation, then the parent control would have to be modified to recognize and use those goals. The hierarchic

control structure of an Epoq optimizer means that the recognition of such new information, and thus any modifications to existing regions, is confined to direct ancestors of a new region.

New strategies that duplicate the goals of existing regions can be added as children to a region without modifying the control strategy of the parent region. Once the parent is informed that the new region exists, the parent region can choose to use it to achieve goals in response to the applicability information and goals provided by the new child region.

5.4 A Formal Basis for Regions

The Epoq approach is based on the idea that regions act as equivalence transformations over queries and can be composed. A region consists of transformations that can be performed on queries and control over the execution of these transformations. In an Epoq hierarchy a region can act as a transformation for another region (its parent). The control strategy of a region defines which transformations can be performed by that region. A region's control over the execution of its subordinates results in a sequence of region executions producing a transformation of a query.

In this section we formalize these ideas. In Section 5.4.1 we define general query transformations. We extend this definition to equivalence transformation in Section 5.4.2. In Section 5.4.3 we characterize control strategies as sequences of equivalence transformations. These ideas form the basis for Section 5.4.4 where we formalize the idea of a region as a transformation.

5.4.1 Query Transformation

Query transformations are partial functions over query expressions. We force these functions to be total by assuming the existence of a null (undefined) query, $\perp$, that can be returned by any function.[9]

In the following, the equal sign (=), when used with queries or expressions, denotes equality of symbols; i.e., $q = q'$ denotes that the symbols q and q' represent identically the same query or syntactically the same query expression.

NOTATION. Let $\mathcal{Q}$ denote the space of all query expressions. We denote an element of $\mathcal{Q}$ as q, q_i, q', etc. The symbol $\perp$ is used to denote an undefined query.

We transform a query expression representing a query q to find an expression representing a query equivalent to q. In the following, we use the terminology 'query' and 'query expression' interchangeably since we assume that a query expression represents a unique query.

DEFINITION 5.1. *A query transformation* $t : \mathcal{Q} \cup \perp \longrightarrow \mathcal{Q} \cup \perp$ *is a function over query expressions. For all t,* $t(\perp) = \perp$.

NOTATION. We use $\mathcal{T}$ to denote the set of all query transformations.

NOTATION. Given $t_i \in \mathcal{T}$ forall i = 0. . .n, $(t_n, t_{n-1}, ..., t_0)$ denotes a finite sequence of transformations. The abbreviation $(t_{n...0})$ represents the same sequence. The sequence () denotes an empty sequence — i.e., a sequence of no transformations.

DEFINITION 5.2. *Given* t_i *in* $\mathcal{T}$ *forall* $i = 0...n$, *and* q *in* $\mathcal{Q}$, $(t_n, t_{n-1}, ..., t_0)(q)$ *is defined to be* $t_n(t_{n-1}(\ldots(t_0(q))\ldots))$, *i.e., the composition of the transformations. The empty sequence applied to q returns q;* $()(q) = q$.

[9] In practice, the input query would be returned instead of $\perp$. However, a success indicator would also be returned by a region, indicating that the result is effectively the undefined query (see Section 6.2). The use of such a result is up to the region requesting the transformation.

OBSERVATION 5.1. *Given transformation sequence* $(t_n, t_{n-1}, ..., t_0)$ *and any* $q \in \mathcal{Q}$, *if for some* $i \in \{0, \ldots, n\}$ $(t_i, t_{i-1}, \ldots, t_0)(q) = \perp$, *then* $(t_n, t_{n-1}, \ldots, t_0)(q) = \perp$.

The processing of a query involves the composition of transformations of the query. The composition of a finite sequence of query transformations is itself a transformation; each transformation produces a query expression which is then processed by the next transformation. If at some point a transformation produces an undefined result, then each subsequent transformation will produce an undefined result since $t(\perp) = \perp$.[10] In other words, a transformation sequence $(t_n, t_{n-1}, ..., t_0)$ is defined over a query expression q only when each t_i is defined for the result of t_{i-1}.

5.4.2 Equivalence transformations

The transformations an optimizer is concerned with are equivalence transformations, i.e., transformations producing expressions that preserve the result of query execution. As discussed in Chapter 3, there is more than one notion of equivalence for queries. These equivalences correspond to different notions of equivalence for query expressions. For the purposes of this discussion we assume some equivalence relation over query expressions and use the symbol $\equiv$ to denote this equivalence.

DEFINITION 5.3. *Given a query expression q and equivalence* $\equiv$, $\mathcal{E}(q,\equiv)$ *is the space of all expressions equivalent to q under* $\equiv$.

NOTATION. We write simply $\mathcal{E}(q)$ when $\equiv$ is clear from the context.

DEFINITION 5.4. $t \in \mathcal{T}$ *is an equivalence transformation over* $\mathcal{Q}$ *when* $\forall q \in \mathcal{Q}$ *if* $t(q) = q'$ *then either* $q' \in \mathcal{E}(q)$ *or* $q' = \perp$.

It is possible that some query transformation t in $\mathcal{T}$ transforms query q into a query in $\mathcal{E}(q)$ but transforms query q_1 into a query that is not in $\mathcal{E}(q_1)$. We do not want to use such transformations in an optimizer.[11] We are concerned here only with transformations that preserve equivalence for all expressions over which they are defined. From here on, unless otherwise noted, the term *transformation* will be used to mean equivalence transformation.

NOTATION. $\mathcal{T}_{\mathcal{E}}$ is the set of all equivalence transformations over $\mathcal{Q}$.

We noted that a sequence of transformations is itself a transformation. Similarly, a sequence of equivalence transformations is an equivalence transformation. Our characterization of the optimization of a query as a sequence of region applications is based on the composability of equivalence transformations.

OBSERVATION 5.2. *If* $t_i \in \mathcal{T}_{\mathcal{E}}$ *for all* $i = 0 \ldots n$, *then* $(t_n, t_{n-1}, \ldots, t_0) \in \mathcal{T}_{\mathcal{E}}$.

Note that the converse of this statement is not true; i.e., if a sequence of transformations is an equivalence transformation we do not know whether the individual transformations are equivalence transformations. We showed in [132], for example, a sequence of transformations that included one weakly equivalent transformation but was overall strongly equivalent — a mixture of two different definitions of equivalence.

[10] Again, in practice, there will be no 'undefined' query and a parent region will have to decide how to proceed with transformations after receiving a 'no success' indicator after the execution of a region. A region could, for example, ignore the unsuccessful transaction and proceed with processing the same query. Or a region could back up to some previous transformation point and try to produce a different sequence of transformations.

[11] It might be interesting to investigate transformations that preserve equivalence over only some subset of queries. A major difficulty here would be in characterizing the input set over which equivalence is maintained. We do not consider such transformations at this time.

5.4.3 Control Strategies

A control strategy, in effect, determines possible sequences of transformations over queries. A particular transformation performed is determined by a region as a function of the query being processed and any previous processing of the region on that query. The potential transformations that can be performed characterize the region's control strategy.

DEFINITION 5.5. *A control strategy $\mathcal{S}$ is a (possibly infinite) set of finite sequences of equivalence transformations. I.e., $\mathcal{S} = \{(t_{i_j}, t_{i_{j-1}}, \ldots, t_{i_0})\}$ such that $\forall i, j\ t_{i_j} \in \mathcal{T}_{\mathcal{E}}$.*

Note that this definition characterizes a control strategy; it does not give a way to build control strategies. However, control strategies need to be implemented in such a way as to satisfy this characterization. For example, a control strategy must terminate processing a query, as indicated by the finite transformation sequences.

Note also that a control strategy includes only equivalence transformations. This is because we are only interested in regions that find equivalent queries.

In general, we would expect the sequences in some control strategy $\mathcal{S}$ to overlap. For example, if the implementation of a control strategy can stop processing a query at any time, then the characterization of the strategy would include sequences that are prefixes of other sequences in the set; i.e. if $(t_i, \ldots, t_0) \in \mathcal{S}$ then $(t_j, \ldots, t_0) \in \mathcal{S}\ \forall j = 0 \ldots i-1$.

A control strategy induces a search graph over $\mathcal{Q} \cup \perp$ in which the nodes are queries and the edges are transformation steps. This graph may be cyclic, since a transformation sequence may, for some queries, result in repeated visitation of a query. These cycles do not retain the information about the finiteness of the transformation sequence.[12] Any query q to which a transformation sequence in $\mathcal{S}$ can be applied is the root of a subgraph of this search graph. The nodes reachable from q using $\mathcal{S}$ are a search space induced by control strategy $\mathcal{S}$.

DEFINITION 5.6. *The search space of $\mathcal{S}$ for query q, denoted $\mathcal{S}(q)$, is the set of queries in $\mathcal{Q}$ reachable from q using the transformation sequences in $\mathcal{S}$. In other words $\mathcal{S}(q) = \{q' \in \mathcal{Q} \mid \exists (t_n, \ldots, t_0) \in \mathcal{S}$ such that $(t_i, \ldots, t_0)(q) = q'$ for some $i \in \{0, \ldots, n\}\}$.*

The search space of strategy $\mathcal{S}$ for a particular query q may be empty. This would occur if the first transformation of every sequence in $\mathcal{S}$ is not defined for q (i.e., if $t_{i_0}(q) = \perp\ \forall (t_i) \in \mathcal{S}$).

OBSERVATION 5.3. *$\mathcal{S}(q)$ is a subset of $\mathcal{E}(q)$.*

Any path in a search graph connects equivalent queries since each edge represents an equivalence transformation. Thus, a control strategy results in a search through the space of queries equivalent to the input query. This matches the characterization of regions in Figure 5.2 — the control strategy of a region limits the search for an equivalent query to a subset of all equivalent queries.

5.4.4 Regions are Transformations

A fundamental characteristic of the control in Epoq is that it uses subordinate regions as query transformers. This ability is based on the idea that a region is essentially a query transformation. The region involves a control strategy for applying transformations and, produces a transformed query result. Thus, we characterize a region by its control strategy and a function that can, for each query, produce an element of the control strategy (i.e., a sequence of transformations) that can be applied to transform the query.

[12] The control over execution of a region implementing a control strategy would, we hope, detect such cycles and adjust accordingly.

DEFINITION 5.7. *A region $\mathcal{R}$ is a pair ($\mathcal{S}$,r) where*

- *$\mathcal{S}$ is a control strategy,*
- *r: $\mathcal{Q} \longrightarrow \mathcal{S} \cup ()$ is a function, called the output function of $\mathcal{R}$.*

The function r characterizes the region's control and indicates that the control responds to the query being processed. Given a query, the region's control implements the control strategy by selecting which transformations to perform, determining when to terminate the application of transformations, and selecting, when possible, a transformed expression that meets the region's goals. These aspects of regions are discussed in Chapter 6.

DEFINITION 5.8. *Given region $\mathcal{R}$= ($\mathcal{S}$,r), the application of region $\mathcal{R}$ to query q, denoted $\mathcal{R}$(q), is defined as r(q)(q).*

In other words, the application of a region to a query is the application of the transformation produced by the region's output function to the query. This definition of region supports our use of a region as a query transformation — the output function produces a transformation sequence that can be applied to transform the region's input query. The definition also supports our characterization of a region as a transformation that applies itself — the application of a region to a query includes the region's application of its output function to the query.

A region may not be able to process an input query; i.e., it is possible that $\mathcal{R}$(q) = $\perp$. This results when $r(q) = (t_{k...0})$ and $(t_{k...0})(q) = \perp$. In practice, we will not let this happen and will require $r(q) = ()$ whenever the control strategy does not define a result for an input query.

In the search graph induced by the control strategy,[13] the result of a region execution will be a leaf of one of the paths in the graph; r produces that path.[14] If r produces an empty sequence, the implication is that q is the only node in a path of length 0. In this case, $\mathcal{R}$(q) = ()(q) = q. Note also that $\mathcal{R}(\perp) = \perp$.

OBSERVATION 5.4. *A region $\mathcal{R}$ is an equivalence transformation over $\mathcal{Q}$.*

This follows from Definitions 5.4, 5.5, 5.7 and 5.8, and forms the basis for the Epoq approach to query optimization. A control strategy can describe sequences of region applications as its transformation sequences. These sequences are well-defined and will search a subset of the queries that are equivalent to the input. The control of a region implements a control strategy by choosing, for each query, a sequence of region applications to transform the query.

5.5 Summary

The Epoq approach to extensible query optimization provides for extending the collection of strategies used for optimizing query expressions. Strategy extensibility is a new kind of extensibility in query optimization that supports any database model but is, in particular, motivated by the requirements of object-oriented database systems.

In this chapter we introduced the notion of strategy extensibility, and presented the Epoq approach to providing this kind of extensibility. An Epoq optimizer is a hierarchy of regions, each of which defines its own strategy to try to achieve some goal on a query expression. The

[13] Again, the graph includes $\perp$ since transformation sequences in the strategy may reach undefined queries.

[14] It is certainly possible that more than one path leads to the same result; we only need to know one of these to be able to find the result.

strategies of sibling regions are integrated through the control of their parent region. We discussed in Section 5.1.3 some of the requirements for providing such a control.

The hierarchic approach of Epoq, and the parent control, are based on a formal view of regions as equivalence transformations that can be composed to optimize a query. The integration of different strategies is supported by common region interfaces, including a common query representation, and by a goal-directed parent control. In the next chapter we present an architecture for the interface and control of regions, and describe a particular control that is based on planning the optimization process. In Chapter 7 we present an extensible representation for queries that can be used by an Epoq optimizer.

Chapter 6

The Epoq Architecture

The Epoq approach to extensible query optimization is to allow extension of the collection of strategies that can be used when optimizing a query. Each strategy can search some portion of the space of queries equivalent to the optimizer input query. Different strategies will usually search different (possibly overlapping) parts of the search space, although different strategies may simply offer alternative ways to search the same space.

In Epoq, a strategy is captured in a region, and regions are combined to process any given query. The major issue addressed by the architecture is how to combine the regions to process a query. In the Epoq architecture, a region is a module, and the modules are connected as a hierarchy. An Epoq optimizer is a rooted, directed acyclic graph of region modules, as shown in Figure 6.1.

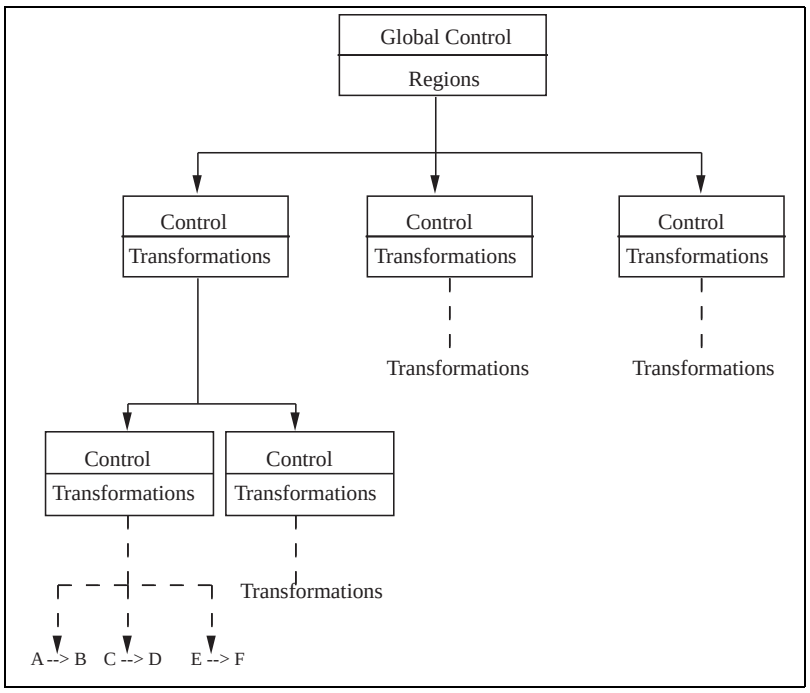

Figure 6.1: Epoq architecture

The root module of the optimizer communicates with the query processing system. It receives a query to optimize, and produces an optimized result. This result is computed with the assistance

of its child regions. Child modules transform queries at the request of a parent region, and may also act as parents by using subordinate regions to assist with this transformation.

Structuring the region modules as a hierarchy puts knowledge about regions that can cooperate to process a single query in one place, i.e. a parent. The parent's strategy determines how its subordinate regions will cooperate. This determination is part of the parent's control over its behavior and the use of its subordinates. In the control proposed in this thesis, a parent causes its children to act in sequence, thereby composing their actions. This control is based on the observation that regions are essentially query transformations – they process an input query to produce an equivalent result query. Since all transformations are equivalence transformations, any ordering of region executions will produce an equivalent query. A parent region composes the transformations of its subordinates to produce such a result query.

Alternative controls, that result perhaps in children processing concurrently, are left for future research. It might be interesting to explore the possibility of different regions working on different parts of a query concurrently, or of different regions processing the same query in parallel with the parent choosing how to combine the different results.

We distinguish two kinds of regions in an Epoq optimizer – interior regions (including the root) and leaf regions. Both kinds of regions are transformations, but they differ in that the control of interior regions manipulates transformations represented by other regions in the optimizer, whereas the control and transformations of a leaf region are internal to the region and are not explicitly addressed in the architecture. Indeed, in Figure 6.1 the leaf regions are shown simply as transformations. Of course, these transformations will in practice be complicated strategies for manipulating queries. The architecture and design presented in this thesis concentrate on the the integration of transformation strategies; we address in particular the control and interface requirements for interior nodes.

An Epoq optimizer can be configured in many different ways. Some of these configurations are depicted in Figure 6.2. Figure 6.2a illustrates an optimizer with a single control over a number of query processing strategies. We expect this will be a common configuration for an optimizer. It is possible, however, for an optimizer to have many levels of regions, and thus levels of control. This is illustrated in Figures 6.2b and c. In Figure 6.2b, two sibling regions share a child region. In Figure 6.2c, the root region and one of its children share a child. As can be seen from this figure, the levels of an optimizer do not have to be distinct. Sharing of children will occur when two distinct processing strategies can both make use of the same transformation. For example, a region processing a nested query and a region that processes only Select operations may want to use the same strategy for manipulating Boolean expressions.

An Epoq configuration will always be a hierarchy; i.e., there are no cyclic region connections. This corresponds to the idea that subordinate regions search a subspace of the space searched by their parent region; each subordinate performs a portion of the overall transformation of its parent. The composition of the transformations performed by subordinates is equivalent to a union of the search spaces.

The Epoq architecture supports the integration of regions, and optimizer strategy extensibility, through a common region interface, a canonical representation for queries, and a hierarchical control over region execution. In the next sections we present these features in more detail. We present an architecture for regions in Section 6.1. The architectural requirements for the region interface and query representation are presented in Section 6.2, and the components of a region's control are presented in Section 6.3. A major component of control is decision-making; in particular, determining which child region will transform a query next. In Section 6.4, we present a control that uses planning to decide the order in which subordinate regions are executed to process a query.

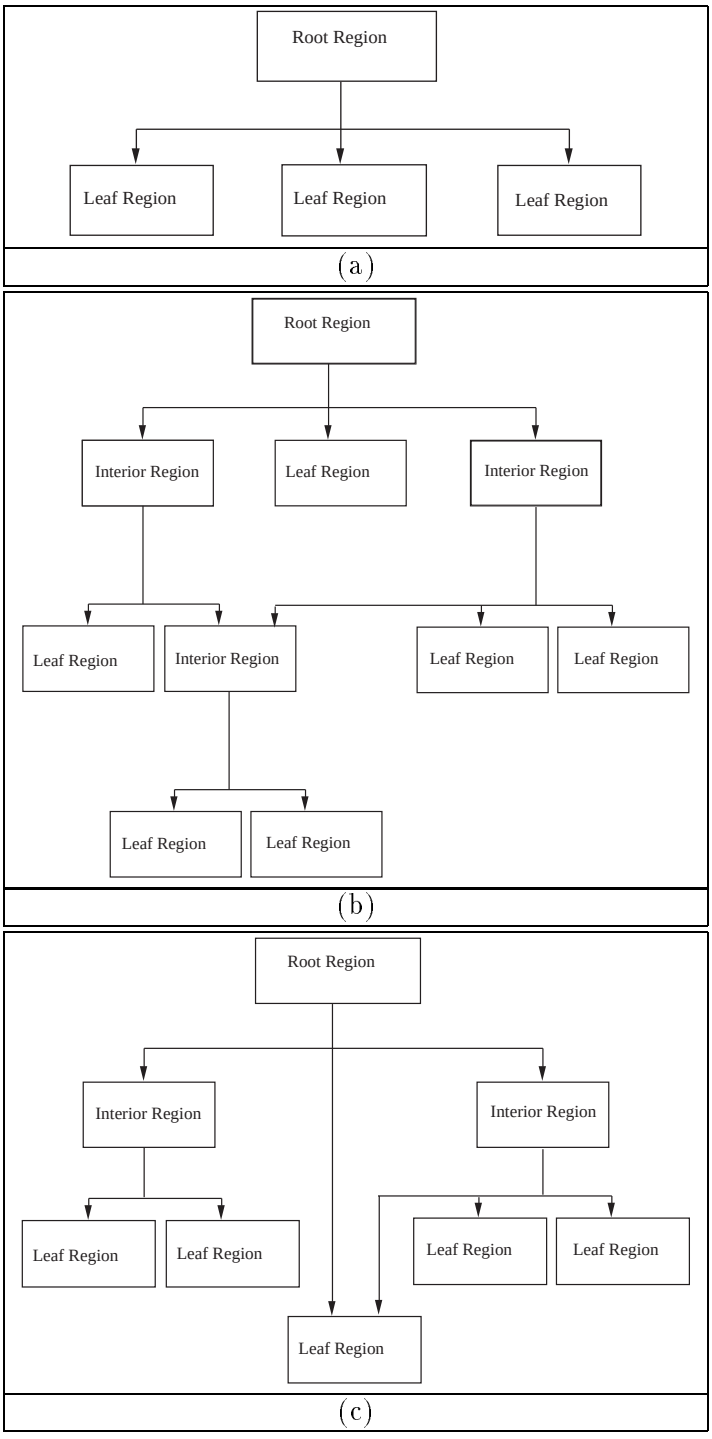

Figure 6.2: Example optimizer configurations

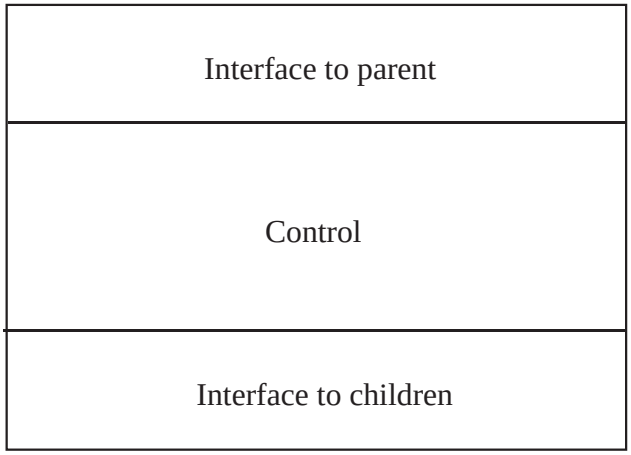

Figure 6.3: Architecture of an Epoq Region.

6.1 Region Architecture

In Section 5.4.4 we showed that the characterization of region as a control strategy and a way of choosing one result of applying that strategy is sufficient to support the idea that a region is a transformation. This is important because, in the Epoq architecture, regions that are children of other regions are used as transformations by their parent.

In Figure 6.1 interior region modules are depicted as control over query transformation, and leaf regions are depicted as simply transformations. In the figure, control indicates that an interior region has control over subordinate regions that do transformations.

All regions, however, have control over the transformation of queries. This conceptual view of a region describes the fact that a region implements a control strategy for manipulating queries. The transformations of a region are those described by the control strategy, and the control is the means by which the control strategy is implemented. For interior regions, the transformations used by the control are subordinate regions as well as, possibly, internally defined transformations. For leaf regions, all transformations are internally defined. Such transformations may be explicit; for example, a rule-based optimizer contains an explicit set of rules and a control mechanism that implements some sort of rule search. The transformations will often be implicit, i.e., built into the control mechanism. For example, a region that uses dynamic programming to generate efficient join orderings uses, implicitly, commutativity and associativity transformations on the join operation.

An architecture that implements this conceptual view of regions is depicted in Figure 6.3. The control of a region provides all the functionalities necessary to implement regions characterized as control over query transformation. These functionalities include deciding which transformations to apply, deciding which query or subquery to process at any point, executing these decisions, manipulating query expressions, determining when to terminate region execution, determining whether the region has successfully transformed an input query, and choosing a result query.

The term *control*, then, refers to the processes that are executed within a region to transform a query. The term *control strategy* is used to describe the conceptual process of query transformation, as in Section 5.4.4. The control strategy of a region is implemented by the region's control.

The region interface provides the support for communication between the control of a region and its parent or child regions. The interface to its parent allows a region to be used as a transformation by the parent. Thus, for example, this interface must at least present a transformation result to the parent. The interface to a region's children allows the region to use the children as transformations.

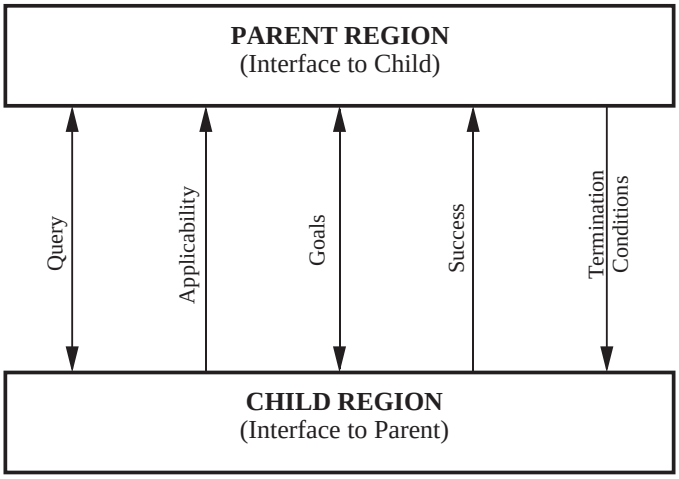

Figure 6.4: Interface Between a Parent and Child Region.

This interface can request information from children which is then supplied to the region's control.

The interface and control components of a region are discussed in more detail in the next sections.

6.2 Interface

A region participates in two interfaces: the interface with its parent(s) and the interface with its children. Leaf regions, of course, have an empty interface to children. Conversely, a parent-child interface exists in two parts: the parent's interface to its children and the child's interface to its parent. The interface of a parent to its children, and that of the children to their parent, obviously must be compatible.

Epoq defines a common structure for the interface to ensure structural compatibility. This supports communication between regions as well as the addition of new regions to an optimizer. As noted in Chapter 5, however, structural compatibility does not ensure semantic compatibility. Although Epoq defines a meaning for the components of the structure, a common meaning for the implementations of these components (i.e., a common meaning for the actual information passed through a parent-child interface) is primarily the responsibility of the system implementor.

The architecture of all parent-child interfaces is the same, although the implementation of this architecture may be different for each interface. For example, the architecture specifies that a query passes between a parent and a child region. Although we would expect all interfaces to use the query representation of Chapter 7, it is possible for a particular parent-child interface to use a different query representation. Of course, if a region R participates as a child in a communication requiring the standard query representation, and R participates as a parent in a communication requiring a different representation, then R is responsible for knowing how to translate between the two representations. The root region, for example, might accept and return a query expression as a string if it can translate that form back and forth between the annotated representation to be used in the optimizer.

The diagram of Figure 6.4 indicates the kinds of information passed between a parent and child region. The arrows in the figure indicate the direction of information flow. For example, queries are passed both ways: a query to be transformed is passed from a parent region to a child, and a transformed query is passed from a child region to its parent.

Goals are also passed both ways: a child can tell its parent(s) which goals it can try to attain, and a parent, when it sends a query to a child region, indicates which goal it wants the child to apply to the query. Success information is passed from the child to the parent to indicate whether the requested goal was actually attained. Goals and success are discussed in more detail in Section 6.2.2.

Applicability information is passed from a child to a parent. This information characterizes the input queries of a child region; i.e., it tells a parent whether or not a child might be able to successfully transform a query. Applicability is discussed further in Section 6.2.3.

Termination conditions can be passed from a parent region to exert control over the execution of a child. Whether or not the child will actually be able to use these conditions depends on the control implemented in the child. The control presented in Section 6.3 can use termination conditions passed by a parent, so we discuss this topic further there.

6.2.1 Static and Dynamic Interface

The interface between a parent and child region can be considered to be two interfaces:

- The *static interface* is query-independent communication. This consists of information shared by a parent and child region without knowing what query is to be processed.
- The *dynamic interface* consists of information passed back and forth between parent and child regions during the optimization of a query.

The term static refers to the fact that the information can be shared before the optimizer executes; similarly, dynamic refers to the execution of the optimizer. Static information may be transferred at optimizer execution time, but the nature of the static information does not require this.

Static information is usually passed from a child region to a parent; it is the means through which a child region tells its parent about its abilities. Static interface information only needs to be provided once to a parent by each subordinate region (unless of course some change in the child region results in a change to the interface information).

Goals and applicability can be provided statically from a child to its parent(s). Applicability describes which queries can be transformed by a region, and a region's goals describe query states that can be attained on queries to which the region is applicable. Both descriptors are basically predicates over query expressions, and it is these predicates that can be statically provided to the parent. A parent can apply the predicates to a query and use the results to help it determine whether to use a region.

Applicability information may also be provided dynamically. In this case a subordinate region would apply its applicability predicate and supply the result to the parent. For example, a region, once it sees the query to be processed and the parent's requested goal, may be able to assess how long it will take to meet the goal. The decision as to whether a particular measure of applicability should be static or dynamic must be made by a system implementor. In general, applicability involving a Boolean assessment of the query alone can be static, but applicability involving assessments of the actual execution of the region (e.g., time or space) should be dynamic.

The query being processed is, by definition, part of the dynamic interface; the query is passed from a parent region to a child and back again after it is transformed. A parent will also pass a goal to a child. This goal must be one of the goals the child can try to attain (the statically defined goals). A region may not be able to achieve its goals for every query it processes, so success information is also part of the dynamic interface.

Finally, the termination conditions requested of a child by a parent are part of the dynamic interface. Such conditions are intended to allow the parent, while doing its own processing, to affect the execution of a child region. This effect is intended to be query dependent. Termination conditions in general are an integral part of a region's control, and should normally be built into that control (statically). The conditions that are passed from a parent may affect the efficiency of a child's control and should therefore be used infrequently. Conditions that are independent of a particular query expression should be implemented integrally with a region rather than being passed through the interface.

6.2.2 Goals

Goals describe query states a region can try to attain. A region defines a set of goals, and those recognized by the parent can be used in the parent control's decision-making. Since a region's statement of goals is independent of the particular query it is processing, those goals are part of the static interface between a region and its parent.

Let $\mathcal{G}$ be the goals domain; i.e. $\mathcal{G}$ is a set of all possible goals. There are two subsets of these goals that participate in a parent-child interface.

- $\mathcal{G}_P$ is the set of goals recognized by a parent region.[1]
- $\mathcal{G}_A$ is the set of goals attainable by a region.

To say a goal is 'recognized' by a parent region means that the parent contains the control mechanisms to request that a child region work towards that goal. A goal is 'attainable' by a region when the region has the control mechanisms to work towards that goal. Attainable does not mean that the region will necessarily achieve the goal for a particular query.

NOTATION. If X is a region, then $\mathcal{G}_P(\mathrm{X})$ is the set of goals recognized by X. Similarly, $\mathcal{G}_A(\mathrm{X})$ is the set of goals attainable by X.

For any set X, $\mathcal{G}_A(\mathrm{X})$ is the set of goals that X passes, through the static interface, to all of its parents. In general, one would expect that, for X a subordinate of Y, $\mathcal{G}_A(X) \subseteq \mathcal{G}_P(Y)$. However, since X could be a subordinate of many regions, it is certainly possible that X could achieve some goals useful to one of its parents, and other goals useful to other parents.

It is important to ensure, however, that the goals of a subordinate intersect the recognizable goals of a parent since the subordinate is useless if it cannot achieve any of its parent's goals. In other words, for regions X and Y, if $\mathcal{G}_A(X) \cap \mathcal{G}_P(Y) = \emptyset$ then X cannot be a subordinate region of Y.

If region Y is a parent of region X, then the goals that can be used by Y in considering whether to use subordinate region X are exactly those goals attainable by X that are recognized by Y. In other words, the only goals that should be passed from a parent to a child are those goals in $\mathcal{G}_P(Y) \cap \mathcal{G}_A(X)$. Any other goals would not be recognized by the control mechanisms of the child.

6.2.3 Region Applicability

Applicability refers to the ability of a region to transform a query. Thus, applicability is directly related to the control strategy of a region; a region for which some transformation applies to a query q is said to be *applicable* to q.

DEFINITION 6.1. *Let region $R = (\mathcal{S}, r)$ and q be any query. R is applicable to q if there exists a transformation sequence $(t_{i\ldots 0})$ in $\mathcal{S}$ such that $(t_{i\ldots 0})(q) = q' \neq \perp$.*

[1] The goals recognized by a parent region are, in Section 6.4, also called primitive goals, thus the notation $\mathcal{G}_P$.

Of course, assessing applicability of a region to a query by finding a transformation sequence is not feasible, so a region needs to use other measures to assess its applicability. We do not require that these measures be exact – they should not, however, eliminate queries the region can transform. An applicability measure for a region should indicate necessary conditions for a region to transform a query.

Practical measures of applicability may consider goals, the input query, termination conditions, transformations, region control, etc. The interface may actually contain two kinds of applicability measures: predicates that indicate whether or not a region believes it can transform a query, and functions that return some comparative measurement of a region's expected ability to transform a query. Applicability measures provide a means for a parent region to eliminate from the decision-making process regions that will not be able to process the current query. Thus, they are used to improve the efficiency of the optimizer.

Static Applicability

Static applicability predicates for a subordinate region describe the form of queries that can be manipulated by that region. For example, a static applicability predicate could describe the fact that a region can process queries containing Join operations, or that a region can process nested queries.

Static applicability is described in a predicate language over queries – we give such a language in Chapter 8. The advantage of describing applicability in a predicate language is that a parent region can determine applicability without actually calling the subordinate region during optimizer execution. The parent can apply the subordinate-supplied predicate to a query, thus reducing the number of calls to subordinate regions. Thus, defining applicability predicates as static is an optional, but recommended, part of the interface.

Dynamic Applicability

Dynamic applicability functions are an assessment by a region of its ability to process a particular query. This assessment may be a more detailed determination of applicability than just indicating that it may transform, or cannot transform, a query. After looking at a query expression, for example, a region might bid for the opportunity to process the query, might provide an estimate of expected cost improvement if the region had the opportunity to process the query, or might provide an estimate of expected processing time.

A parent region requests, while the optimizer is executing, that a subordinate evaluate dynamic applicability functions. The parent is assessed (statically) of the applicability functions available to be executed by the subordinate and will provide input parameters for functions when it requests their evaluation by a subordinate. For example, a function that bids for the opportunity to process a query might have type Query $\rightarrow$ 1..10, where a bid of 10 indicates that the region is very eager to process the given query. A parent region could send a query to be processed to each of its subordinates and ask them to bid on the query. The function results are assumed to be consistent across regions. This could be assisted by semantic information provided to region-builders, and could be more strictly enforced by normalization of region responses by the parent as discussed in Section 5.1.2.

Although it is possible that a subordinate could provide dynamic applicability functions to the parent for the parent to execute directly (as is done for static applicability predicates), it is expected that assessments of dynamic applicability will be so closely associated with the subordinate's control that it will be more efficient for the subordinate itself to execute such functions. Indeed, dynamic

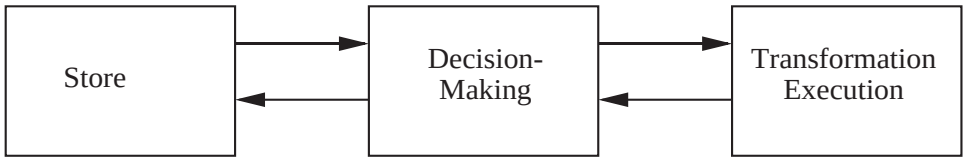

Figure 6.5: Components of Region Control

assessment of applicability may involve information that is internal to a region and should not be exposed (except implicitly through the result) to the parent.

Summary

Applicability in an Epoq optimizer is similar in function to pattern-matching and condition-matching of left-hand sides in more traditional rule-based optimizers. Applicability differs in that it doesn't guarantee that a region can process a query, but only tells when a region can not process a query. This difference is accounted for in the region control. A region requests that an applicable subordinate achieve a particular goal on a query, and must be able to accommodate failure of the subordinate to achieve that goal.

6.3 Control Architecture

The actual transformation of a query expression is the responsibility of the region control. The region interface provides information to the control, and relays control results and requests to other regions, and the control components interact with each other, and with the interface, to effect the transformation process. Through its management of the transformation process, the control implements a control strategy for the region.

The control of an interior region uses subordinate regions to perform its transformations, while the control of a leaf region performs all query transformations internally. In either case, however, the control consists of the same basic components. These components will be implemented differently for different regions, resulting in different control strategies for the regions.

In this section we discuss the components that are present in all region controls. In Section 6.4 we look more specifically at an extensible interior control that is based on planning the optimizer execution. The interior control is concerned, in particular, with determining sequences of subordinate region executions that transform a query to meet the region's goal.

6.3.1 Overview

The major components of a region control are pictured in Figure 6.5. The decision-making component is the central part of the region control. This component makes the decisions about what processing the region must do to transform a query. It interacts with the control store to obtain information used in making decisions and to store information that may be used in later decisions. The decision-making component decides what transformations should be performed (which, for interior regions, translates to which subordinate regions should be executed) to manipulate a query. Conversely, the results of transformations can be input to decisions about how to (and whether to) continue the transformation process.

A high-level view of region execution is depicted in Figure 6.6. Basically, a region continues transforming a query until the decision to terminate the processing is made. A region first chooses

```
initialize store
while termination conditions not met do
   choose a query and goal
   choose a transformation to process
   transform query
   update store with result info
endwhile
return "best" query and success indicator
```

Figure 6.6: Basic Region Execution

a query or subquery to process and a goal for that processing. An initial query and goal are provided by a parent region through the interface, but as processing continues the region may work on intermediate results, or on subqueries, with different goals in order to try to attain the required goal on the initial query. Intermediate transformation results may be stored, and after termination a result to be produced by the region is chosen. That result may, or may not, reflect success at achieving the initial goal for the transformation process.

6.3.2 Control Store

The control store supports decision-making by maintaining information used during region execution. Items in the store are named (e.g., variable names) and typed so that they can be directly accessed by the region control and can be manipulated by methods defined over their type. The control state consists of the contents of the store, along with information about the current processing step.

The store always includes an initial and current version of the query being transformed, as well as intermediate (equivalent) versions of the query. Intermediate query versions provide choices of ways to execute a particular query, and also provide processing choices for an optimizer. For example, in a rule-based optimizer intermediate versions of a query or subquery are usually maintained and the rule search engine (decision-making component) will match rules to these versions for further processing.

In order to support the control decisions, the store will also include a log — i.e., a history of previous processing. The log is used to keep track of the actions performed during the processing of queries. Such information could be used in control decision-making, for example, to prevent repetitive processing. It can also be used to support the undoing of partial results after failure to attain a goal.

The store may also include other data as defined and required by a particular control. For example, the control defined in Section 6.4 can define a working memory similar to that defined for rule-based programming languages such as OPS5 [34]. The memory can then include information that helps direct the control execution.

A region's store is local to the region. Any information that should be visible to other regions is passed as arguments through the region interface. The store is initialized by information passed at the interface when the region is called by a parent. The current query and goal information are stored, and the log is initialized, to indicate the parent's request. If a parent sends termination conditions, these must also be stored.

Get_Cost:	Query	$\longrightarrow$	cost
Get_Equiv:	Query	$\longrightarrow$	QuerySet
Choose_Best:	QuerySet, Goal	$\longrightarrow$	Query
Prune:	QuerySet	$\longrightarrow$	QuerySet
Add_Choice:	QuerySet, Query	$\longrightarrow$	QuerySet

Figure 6.7: Methods supporting decisions and queries.

In general, a region's control store persists for the duration of the region execution. When the region terminates processing, a result is produced and the region memory disappears. There are, however, circumstances under which some portions of control state must persist through different region executions. In particular, a control might want to maintain statistical information about its processing in order to adapt future processing decisions. Such information would have to persist after a region completes a particular execution.

Query Storage and Management

The store will contain queries, and two relationships between these queries: subquery and equivalence. The subquery relationship is an explicit part of the query — arguments to a query can be other queries. Equivalence relationships are instances of a type called QuerySet. The instances of this type are, abstractly, sets of queries; all queries in a single queryset are equivalent to each other. The store will always contain at least one queryset — the set of queries equivalent to the initial query.

Some methods for supporting decisions about queries and querysets are shown in Figure 6.7. Methods Get_Cost and Get_Equiv retrieve information about the state of a query. The Get_Cost method retrieves the current cost of a query. That cost will depend on the cost model being used in the optimizer. Retrieving the cost is supported by methods of the cost model. The Get_Equiv method returns the queryset associated with the input query parameter. This gives access to the queries that are equivalent to the input query, thus allowing further processing such as iterating through the equivalent queries, or choosing an equivalent query for processing.

Methods defined for Type QuerySet may involve decision-making. For example, a queryset supports method Choose_Best to find the query in a set of alternative queries that best achieves the required goal. Add_Choice is used in the control whenever a new query is generated through transformation. The region control can decide, through the implementation of Add_Choice, whether or not a particular transformation result is stored for future manipulation. Method Prune further manages querysets by removing equivalent queries that are no longer useful to the control.

The Query and QuerySet types are global to an optimizer, but the type representations and method implementations may be redefined within a region. Some methods would be expected to be implemented in the same way for all regions. For example, updating cost information would be the same for all regions.[2]

Most methods would be re-implemented in a region, however, because the implementations of the methods help define the control strategy of the region. For example, if a region processes a query as, simply, a sequence of transformed expressions then the representation could store only the most recent query version. In this case, the Add_Choice routine would replace the previous

[2]Although, if a region maintains its own internal cost model, it may need routines for cost update relative to that model as well as routines for updating cost relative to the model at the interface.

query with the new choice, the Prune routine would be null, and the Choose_Best routine would simply return the query. A region may also define additional methods for query manipulation. For example, a region may have a routine that can detect common subexpressions when processing a query.

Log Management

The log is a record of actions taken in a region. It includes a record of query modifications (i.e., transformations) and may include a record of modifications to other information stored in memory. The log can support control decisions by recording the results of previous decisions. It can also support a simple recovery scheme by offering the ability to restore the memory to a point prior to some transformation of a query.

The log must support a method Append(Log, LogRecord) to add a new record to the log. The Retrieve actions supported by a log depend on the requirements of the particular decision-making component, and could include retrievals to answer such queries as:

- what was the last transformation performed?
- what was the last transformation performed on query Q?
- what was the last transformation that resulted in query Q?
- has query Q been transformed yet?
- has a particular transformation been used?

6.3.3 Decision-Making

The primary component of a region is the decision-making component. The decisions that are made in a region determine the transformations that will be performed on a query, the modifications that will be made to the store, and future decisions that will be made.

The decision-making functionality consists of components to choose a focus for processing (i.e., a query and goal), choose a transformation to execute, decide what to record in the store as a result of a transformation, decide when to terminate processing, and determine the result of the region's execution. These decision "modules" are reflected in the execution depicted in Figure 6.6.

The region focuses on a particular query and goal, and chooses a process for transforming the query. These two decisions are complementary — the transformation to perform depends on the query and goal, and the query and goal chosen may depend on what transformations are available. This duality is evident in rule-based optimizers, where queries are matched to transformation rules and a best query/rule pair is chosen for next execution. In Epoq these decisions are made by a planning system. The query and goal form a task for a region to perform, and the planning system finds a sequence of transformations that tries to perform the required task. This system is described in detail in Section 6.4.

The decisions about modifications to make to the store can be incorporated in the implementations of methods that update elements in the store. For example, the procedure call

Add_Choice(Get_Equiv(input query), result query)

requests that the result of a transformation be recorded as a query that is equivalent to the query transformed (i.e., input query). Add_Choice could be implemented to decide whether the result query is sufficiently 'good' to store. On the other hand, Add_Choice could be implemented to

always add a new query and there could be a separate decision-making method that determines whether a result should be added.

The final decision made by a region is the choice of query result. This decision is related to the processes that store alternative queries since the final query will be one of the stored alternatives. It is also related to a Success? method for the region goal, since a region will choose (if possible) to return an equivalent query that meets the goal. A region must return a query that is equivalent to the input query, thus if no satisfactory alternative is stored the region returns the original query.

Termination

Termination refers to the conditions under which the region stops its processing and returns to its parent. These conditions are checked after each query transformation (see Figure 6.6) and, if the conditions are met, processing is completed.

In general, the conditions for termination are built into the control. For example, a control may decide to terminate after processing a certain number of transformations, after finding a particular number of alternative queries, after generating an alternative that attains the region's goal, after finding no good prospects for further processing the input query, etc.

However, the hierarchical organization of region processing means that a parent region may want (or need) to have some control over the termination of its subordinate regions. Such control can be implemented by sending termination conditions to a child region through the parent-child interface. The disjunction of the built-in conditions and the parent's condition then becomes the termination criteria for the region.

A parent can send two kinds of termination conditions: conditions involving the state of the query and conditions involving the utilization of resources. The former can be submitted to a subordinate as Boolean functions with one parameter — a query. At each termination check, the child can supply the query argument to the parent's termination function and execute the function. For example, suppose a parent will be satisfied if a subordinate that works to improve cost can improve the cost of the query by 15%. The parent would supply a Boolean function *Improve(initial query, current query)* that computes the percentage cost difference between an initial query state and the current query state and returns true if that percentage is greater than or equal to fifteen. The parent would send a closure of Improve, with the initial query instantiated, to its child through the interface. At each check for termination, the child would supply the current query and execute the function.

Termination conditions involving resource utilization generally require reasoning on the part of the child region. As a result, if a region wishes to convey such conditions to its children, the children must be constructed to directly respond to the particular conditions. For example, suppose a parent wishes to set time limits on a child's execution. The child must be able to interpret the time limits and, if it in turn calls other regions, must be able to apportion time limits to its children.

6.3.4 Transformation Execution

Transforming queries is done in response to decisions made by the control. For leaf regions the transformations are done internally. For interior regions a query transformation usually involves invocation of a child region. Thus, the transformation execution module is the mediator between control decision-making and the parent-child interface.

To invoke a child region, the query, goal, and termination condition parameters must be instantiated. The goal for execution can be passed directly to the child, but a copy of the query must be sent as the query argument. The copy semantics for the query argument reflect the fact that

the original query and intermediate transformations are maintained in the store. The child region should not directly manipulate the parent's store, and thus must manipulate a copy of the desired query. Any modifications to the copy are returned to the parent, where the decision about storing the modified query is made.

The execution of a transformation is subject to conditions determining when the transformation should terminate. As noted in the previous section, global termination conditions are passed between regions through the interface, thus the transformation execution module must be able to prepare such arguments. Such conditions may need to be modified as they pass from parent to child. For example, conditions about resource utilization might be modified to reflect the resources that have already been used by the parent region.

The execution of a transformation will result in a log entry indicating the query and goal for transformation, the transformation performed, and the result. The log entry can reference the store to indicate the query that is input to the transformation, and will also reference the store if the result is chosen to reside there. If the result is not chosen to be stored, that fact will be noted in the log. The transformation information in the log entry includes an indicator (i.e., name) for the region performing the transformation. This log information forms a transformation history that can be used by the decision-making system when planning future actions. The query state information in the log could also be used to recover the query information in the store if, for example, the decision system wants to back up and restart a sequence of transformations.

6.4 A Planning-Based Control over Regions

The Epoq architecture is based on the assumption that a region can determine a sequence of executions for subordinate regions that will result in a desired query transformation. In this section we present a region control that justifies that assumption. This control is based on planning the optimization process.

Planning reduces a task to primitive actions [28]. In the Epoq context, a region's task is to achieve some goal on a query, and primitive actions are subordinate region executions. Plans describe how primitive actions can be combined to achieve a task.

Plans use goals to describe tasks and subtasks that can be performed to achieve other goals. The planning process uses goals to progressively break a task into subtasks, and eventually to primitive tasks (i.e., region executions). Thus the result of planning is a sequence of subordinate region executions that may achieve the region's goal.

Plans are defined in a rule-based language, where the rules provide heuristics about potentially good interactions between region executions. A rule search engine matches rules to the current goal and control state (including the current query), and chooses rules to execute. Rules describe three kinds of actions that can be performed: planning actions, primitive actions, and memory updates. Primitive actions result in subordinate region executions. Planning actions induce a forward chain through rule goals to find primitive actions. Update actions modify the region's working memory with information that may be used later in the rule search.

The rule language and working memory are patterned after rule-based languages such as OPS5 [34]. A major difference, though, is that the Epoq rule interpreter interleaves rule manipulation with the execution of subordinate regions. This interleaving of planning and execution is required because subordinate regions can fail to achieve their stated goal, and the success or failure of a region execution to achieve its goal affects the optimization process. In other words, the sequence of actions that is generated as a plan is affected by the execution of those actions. This is similar to reactive planning in the domain of robot control [46, 52, 76].

In the remainder of this section we present the components of the planning-based control in more detail. The working memory stores control state information and is discussed in Section 6.4.1. The rule language is presented in Section 6.4.2. Rules describe tasks that can achieve a particular goal. Rules that achieve the same goal are collected into goal packages, discussed in Section 6.4.3. In Section 6.4.4 we present the execution model for the planning system, and in Section 6.4.5 we examine how failure of a task to achieve its goal is handled in this model. In Section 6.4.6 we discuss the inherent extensibility of the planning-based control.

6.4.1 Working Memory

The working memory stores the control state of a region. It consists of (name, object) pairs where each object is an instance of some abstract data type, and the name is an index that can be used to retrieve the object from the store. Working memory will store objects such as the current query being processed or alternative queries that have been discovered through transformations. In Epoq, the current and alternative queries can be stored using the query representation presented in Chapter 7 and are manipulated by the query manipulation methods of Figure 6.7. A log of region actions is also maintained in working memory. This is discussed in more detail in Section 6.4.5.

Abstract data types can also be used to define tuple objects, such as those used in OPS5 [34]. Such objects would be used to help control rule execution. The working memory elements can be matched in rules and, in that way, can provide information referenced in rule actions as well as information that helps control the selection of rules to execute.

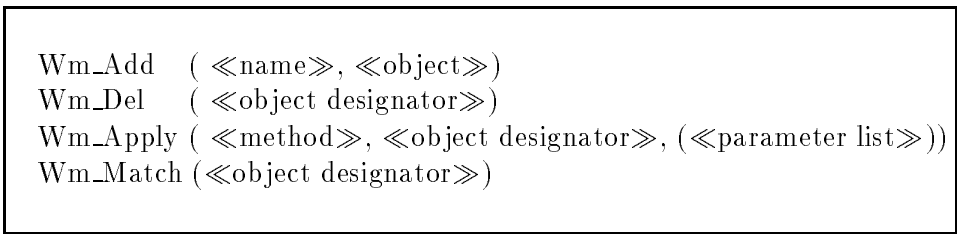

Figure 6.8: Methods over working memory.

The collection of working memory elements can be accessed using the methods shown in Figure 6.8. The Wm_Add method adds a (name, object) to the working memory. The object parameter might be a variable referencing a currently existing object, but could also be a function returning an object. For example, suppose the method **new** on type Flag requires an integer parameter called "id" and a query parameter called "query". The method application

Wm_Add(Done, (Flag new(id = 1, query = Q)))

adds a flag named Done to working memory.

Wm_Del removes the object(s) designated by its parameter from working memory. An object designator gives a type and either a name for an object – a variable reference or a working memory name – or a predicate describing an object (discussed later). For example, Wm_Del(Flag Done) deletes the flag added in the last paragraph (and any other flags that happen to have the same name).

Wm_Apply is used to apply methods to working memory objects. These methods must be defined over the type of the object. The parameters required are a method to be applied, a way to

determine the object(s) to which it is applied (i.e., an object designator), and a list (optional) of any other arguments to the method. For example,

Wm_Apply(Update_Cost, (Query Current_Query))

is a directive to apply the Update_Cost method to the query named Current_Query. Such a directive might be used, for example, as the last step of a rule to ensure that the query cost is up-to-date before the next rule is chosen and executed. If a method returns a result, Wm_Apply will forward the result. For example,

Wm_Apply(Choose_Best, (QuerySet QS), (G))

applies the Choose_Best method to the queryset denoted by QS, with a goal argument G. The query result of Choose_Best is returned by Wm_Apply.

These working memory methods are actions that can be performed by planning rules. Rules can also use the Boolean operation Wm_Match to pose queries over objects in working memory. The operation returns True if it finds an object in working memory matching the object designator. As noted earlier, the object designator may be a type and a name for an object. However, the designator will usually be expressed using typed predicates such as those defined for the EQUAL Select operation. Such predicates are Boolean expressions whose terms can query the information at the interface of the object. For example,

Wm_Match(Flag id = 1)

returns True if there is an object of type flag in working memory whose **id** method returns the value 1.

We augment the predicate language by allowing the inclusion of unbound variables. These variables can be used to return information about the matched object. Wm_Match will try to match an object on all bound terms, and when finding a match will further bind variable terms to values of the matched object. For example,

Wm_Match(Flag id = 1 $\wedge$ query = ?Q)

searches for a flag whose id = 1 and binds the query property of the flag to variable ?Q. This augmented matching can be useful for passing information between rules, as will be seen in the example of Chapter 8. It is patterned after the condition elements of OPS5 [34] extended to work with abstract objects.

This style of memory predicate could also be used, for example, to test for subqueries of a query and match subqueries to variables that will be referenced in actions on the rule's right hand side. This is a promising way to deal with subqueries that we choose not to fully explore in this thesis.

6.4.2 Rules for Planning

The planning process uses rules that describe heuristics for good interactions between optimization tasks. These heuristics guide the decision-making process.

Planning rules have the general form

$$\textit{condition test} \longrightarrow \textit{action sequence}$$

with the semantics that *if* the left hand side condition is satisfied, *then* the actions on the right hand side are executed in sequence. The left hand side conditions are predicates over working memory

and applicability predicates over the current query. The right hand side actions are planning or primitive actions, or memory updates.

Applicability conditions for a region describe query states that can be manipulated by the region; the predicates in a rule further specify the queries to which the particular rule is applicable. For example, suppose a region specifies that it can process any query. One of the rules in the region may specify, through applicability predicates on its left hand side, that it is a heuristic for processing queries with only Select, Project or Join operations.

The condition test of a rule is a Boolean combination of

- applicability predicates over a query
- Wm_Match operations
- any Boolean expression all of whose variables are bound

Conditions are tested from left to right, so bindings made in one condition can carry to subsequent clauses. For example, suppose variable Q is bound to a query. The condition test

$$\text{Wm\_Match}(\text{Flag id} = 1 \wedge \text{query} = ?\text{Q}) \wedge (\text{est\_cost}(?\text{Q}) < \text{est\_cost}(\text{Q}))$$

will search working memory for a Flag with id=1 and, if found, will bind ?Q to the query property of the flag. The next clause can then compare the cost of ?Q with that of query Q. Such information could be used by a rule, for example, to determine whether to process query Q or ?Q.

Rules also have an implied predicate that tests the current goal of the region. This predicate is implemented by collecting rules that test for the same goal into *goal packages*. In other words, all rules in a goal package describe ways to achieve the same goal. The applicability and working memory predicates are the factors that allow the planning system to distinguish between the rules.

The right hand side of a rule describes a sequence of steps, or subtasks, that should be taken to attain the desired goal. The steps are either goal actions or memory updates. A goal action has the form

ACHIEVE ≪goal_index≫ ON ≪query_variable≫ [GIVING ≪query_variable≫]

indicating that the goal identified by ≪goal_index≫ be achieved on the query represented by the variable in the ON clause. The GIVING clause is optional and specifies a new variable for storing the transformation result. If no GIVING clause is used, the input query is modified.

Achieve actions describe subtasks that need to be accomplished to attain the rule goal. The goal can be a primitive goal that will be attained by the execution of a subordinate region, or a planning goal described with a goal package. Planning goals elicit a forward chain through goal packages, searching for primitive goals to execute.

For example, consider the following rule:

nested(Q) $\longrightarrow$ ACHIEVE Flatter ON Q;
ACHIEVE Join_Reorder ON Q.

The predicate *nested(Q)* is an applicability predicate that checks for nested queries. If Q is nested, the rule indicates that two actions should be taken in sequence. The first action achieves goal Flatter. If Flatter is a planning goal, this action will initiate further planning for that goal. If Flatter is a primitive goal, a subordinate region will be chosen to attain the goal. Similarly, Join_Reorder could represent a planning or primitive goal. The result of achieving the Flatter goal on Q is input to the Join_Reorder task. Join_Reorder updates Q, completing the actions of the rule.

GOAL PACKAGE	≪goal name ≫
QUERY	≪query variable ≫: default Q
SUCCESS	≪success function ≫: default Success?(≪goal name ≫)
SEARCH	≪search method ≫
TERMINATION	≪termination conditions ≫
LOCAL STORE	≪type or object specifications ≫
RESULT	≪query variable ≫: default QUERY
METHODS	≪list of rules ≫

Figure 6.9: Goal Package Layout.

Goals

We model goals as instances (objects) of type Goal. The type defines that an instance has a unique name and a Boolean Success? function that encapsulates the criteria for goal success. The Success? method has one query parameter for which it tests the goal. Type Goal can be subtyped to allow redefinition of the Success? method, in particular to include additional parameters. For example, a comparative goal would have a Success? method with two query parameters – an initial query and result query – that can be compared to determine success.

Recall (from Section 6.2.2) that $\mathcal{G}$ is used to denote all goals in an optimizer. Let $\mathcal{G}(\mathrm{R})$ denote the goals that region R uses in its decision-making process; i.e., the goals used in any of the planning rules of region R. For any of the goals in $\mathcal{G}(\mathrm{R})$ there must be either rules describing how to attain the goal or subordinate regions that can attain the goal. The set $\mathcal{G}_P(\mathrm{R})$ ($\subseteq \mathcal{G}(\mathrm{R})$) is the set of primitive goals of region R; i.e., the goals a region expects to be attained by its subordinate regions. The set $\mathcal{G}_A(\mathrm{R})$ ($\subseteq \mathcal{G}(\mathrm{R})$) is the set of goals that a region, as a child, can attain for its parents. Both sets of goals are revealed at the region interface, as described in Section 6.2.2.

6.4.3 Goal Packages

Rules describing ways to achieve a particular goal are collected into goal packages. There must be a goal package defined for each goal in $\mathcal{G}(\mathrm{R}) - \mathcal{G}_P(\mathrm{R})$.

Goal packages modularize the planning process in the same way that regions modularize the optimization process. Collecting rules with the same goals into a package allows for the definition of package search strategies that can take advantage of the smaller sets of rules and of any particular characteristics of the rule sets [115, 137]. Each goal package has its own execution and private control store. Thus, as for regions, different goal packages can define different search control strategies and termination conditions.

The local control store of a goal package is manipulated by operations analogous to those for global working memory. We prefix the operations with Lwm (instead of Wm, e.g. Lwm_Add). The local control store is private to the package; it is not available to any packages or execution modules called from the package. Any communication with other packages is done through the interface or, if necessary, through the global control store.

A goal package has the form shown in Figure 6.9. The GOAL PACKAGE clause gives the name of the goal of this package. The SUCCESS clause may default to the success function provided with the goal object, or may specify a specialized success function. For example, a package with a

GOAL PACKAGE Lower_Cost QUERY Q
SEARCH give priority to most matching clauses ∧ one success per rule TERMINATION no rule applies
LOCAL STORE OBJ Q_0, Q_1: Query ← Q RESULT Q ← choose lower est cost (Q_0, Q_1)
METHODS NOT EMPTY (Preds(Q)) ⟶ ACHIEVE Predicate_Reorder ON Q GIVING Q_0; ACHIEVE Join_Reorder ON Q_0. ⟶ ACHIEVE Best_Est_Cost ON Q GIVING Q_1.

Figure 6.10: Example goal package.

goal of Join_Reorder may be able to determine success by whether or not a rule which reorders a query is successful. Thus, it would not need to use the Success? method of the goal. This could be more efficient; e.g., the Success? method for the goal might be expensive if it had to search through the query representation.

The QUERY and RESULT clauses give variable names for the input and output query, respectively. The QUERY variable is used to reference the query as input to rules. The default QUERY variable is Q, and the default result is the input variable.

The LOCAL STORE declarations are types and variables that may be used in rules. Types are declared in a TYPE clause, and variables are declared in an OBJ clause. The OBJ clause must give a variable name, and a type for the variable.

The METHODS section lists the rules of the package. The SEARCH and TERMINATION clauses specify parameters for the goal package execution. SEARCH may specify such parameters as "one success per rule" with the meaning that after a rule has been successfully applied to a query it is no longer available to be applied to a query, or "priority by rule number" meaning that a default rule priority is by the position of the rule in the METHODS list. TERMINATION specifications are conditions for ending the iteration through rules in the package.

For example, consider the goal package of Figure 6.10. The goal of this package is Lower_Cost, and success for the package is Success? for the goal. The input query is manipulated as query Q, and other queries used in rules are specified in the local store (Q_0 and Q_1). The two locally specified query variables are initialized with the value of Q, and may receive new values as a result of rule applications. The result query is computed by choosing, of queries Q_0 and Q_1, the query with the lower estimated cost.

The SEARCH clause specifies that the search engine, if it finds that more than one rule applies, should choose the rule with the most matching clauses.[3] In addition, the search engine cannot consider a rule that has already been successfully applied. The region execution terminates when there are no rules to apply. Termination is guaranteed since the "one success per rule" condition decreases the size of the rule set at each iteration of the package.

In general, goal packages capture rule iteration. At each iteration step, a rule is chosen to apply to the current query. If the rule completes its execution, the iteration continues. If the rule doesn't

[3]Giving priority to most matching clauses implies a count of the clauses (i.e., simple predicates) applied to satisfy a rule condition. Rules with no condition (such as the last rule in the Lower_Cost package) have a count of 0 and thus will have lower priority than other rules. Rules with disjunctive conditions will have disjuncts satisfied from left to right, so the count will indicate the number of clauses satisfied until the entire condition is found to be true.

complete (for example, some step of the rule fails) then an alternate rule can be chosen to try to accomplish the same task. For example, the simple example of Figure 6.10 will, at most, iterate twice – applying one rule each time. If both rules apply, the first rule will be tried first. If it fails, the second rule will be applied. The failure of a both rules will terminate the package execution.

Goal packages can also model conditional behavior. The execution of a package chooses a rule based on the conditions expressed on the left hand side of the rule. Thus, one use for a goal package is to modularize a collection of rules with disjoint conditions, implement a sequential search through the rules, and choose the first rule whose conditions are met.

6.4.4 Region Execution

The desired execution model for the region's planning system is eager in the sense that a single rule (i.e., task) is executed to completion before trying any other rules. The Achieve actions of the rule induce either a forward chain through rule packages or a region execution; Update actions modify working memory. The actions of a rule are performed in sequence, so the execution will chain through rule packages until a primitive goal is encountered. Primitive goals are immediately executed, then processing is returned to the next step in the rule that invoked the primitive goal action. When a rule successfully completes, the newly transformed query is stored and processing may continue with another query, goal and rule. If a rule fails, another rule with the same goal will be tried.

In order to effect this execution model, processing is divided into the following modules:

- High-level region execution — interacts with the interface modules; searches for queries, or subqueries, and goals to process; executes goal packages
- Package execution — chooses rules to achieve the required goal
- Rule Execution — processes, in sequence, the actions of a rule
  - Achieve action — invokes processing to achieve the stated goal
  - Update — updates working memory
- Primitive Action — executes subordinate region

High-level region execution begins execution of the first package and continues execution until region termination conditions are met. Package execution continues executing rules to achieve the package goal until its termination conditions hold. The combination of these two execution modules is basically a rule search system.

Search termination conditions that can be passed from a parent are called *global termination conditions*. Any global termination conditions (as discussed in Section 6.3.3) are combined with the built-in conditions in both the high-level and package execution loops.

Global termination conditions are also checked by primitive goal actions before executing a subordinate region. If global termination is satisfied, the primitive action will not execute the subordinate and will fail. This failure prompts a failure of the rule, which returns control to package execution. At this point the global termination will again be detected and control returned to the high-level region execution module which will terminate the region processing.

Details of the execution modules are given in the following subsections.

```
INPUT: query Q, goal G, termination conditions
OUTPUT: query, Boolean (success indicator)

initialize current goal, current query, global termination conditions
while ≪termination conditions≫ not met do
    choose current goal and query
    invoke package for goal
endwhile
result ← Choose_Best(Get_Equiv(Q), G)
return result and Success?(Result)
```

Figure 6.11: High-level region Execution

High-level region execution

The invocation of a region passes information needed for the region execution through the parent-child interface and begins execution of the control loop depicted in Figure 6.11. This loop, along with the nested loop for package execution, effects a search through the rules that guide the optimization process for rules that match the current query and control state. Initially, a region is given a query and a goal to achieve on that query. The high-level execution module can decide to work on that query, or may decide to process a subquery. As a query is processed, transformations of the query create alternative queries that may be chosen for processing in later executions of the high-level loop.

Termination conditions for the loop consist of built-in as well as the global conditions. These conditions are combined such that either the built-in conditions or the global conditions can terminate processing.

Package execution

Goal packages are collections of rules that can attain the same goal and, as a result, the combination of high-level and package execution is analogous to rule search. The high-level execution module determines what goal will be pursued, and package execution uses additional conditions on rules to find rules to achieve the goal.

The main job of package execution is to ensure that rules are executed to completion. A rule that completes may achieve the required goal, but a rule that doesn't complete will not achieve the goal. Thus, package execution will try rules that are applicable to a query until a rule executes to completion.

A single rule application may meet the package's goal, but the rules in a goal package may also be applied iteratively to work towards the goal. For example, a goal of lower cost may actually be achieved by successively lowering the cost of the query until a satisfactory result is obtained. The termination conditions of the package determine the amount of iteration necessary to achieve the goal.

In order to find a rule that may be successful, the package execution module uses the SEARCH parameters indicated for the package, and conditions on the left-hand sides of rules, to determine a priority ordering for the rules. After a rule is applied, the module checks to see if the rule executed to completion. If the rule completes, termination conditions are used to determine whether the

```
INPUT: query Q
OUTPUT: query, success indicator

Q_save ← Q
set up a priority queue of rules whose conditions are met
while (queue not empty) ∧ (≪termination conditions≫ not met) do
        execute first rule in queue on Q
        if rule completes
            reinitialize priority queue of rules
        otherwise
            remove first rule from queue
    endwhile
    return RESULT and SUCCESS
```

Figure 6.12: Package Execution

package will further manipulate the result. If the rule doesn't complete other rules are tried, in the priority order, until a rule is successful or there are no more rules that can be applied to the query.

Rule priorities can be determined in a number of standard ways. First of all, the conditions on the left-hand sides of rules may eliminate rules from consideration. Of the rules that remain, priorities could be determined by prior assignment of priorities to rules (e.g., the first rule in the package has highest priority), by the number of matching conditions in a rule, by assigning priorities (such as time) to the working memory elements that are matched to rule conditions, or randomly (see [34] for example). These priorities are indicated in the SEARCH clause for the package.

One built-in termination condition for a rule package is that all rules have been tried. The 'empty queue' condition reflects this termination condition. Other termination conditions for a package are determined by the requirements of the package goal. As noted in Section 5.1.3, termination may be related to success at achieving the goal but must also have conditions that are independent of success. In addition, region termination conditions are combined with any global termination conditions to ensure that no more rules are attempted when global termination is indicated.

Rule execution

The required execution is that a rule runs to completion before any further rules are executed. This is a major difference between the Epoq rule engine and most other rule systems. In Epoq we require sequential execution of rule steps, with no intervening rule execution. A major motivation for this control is the interaction between the results of execution of subordinate regions and the planning system. In particular, the fact that subordinate regions and therefore, eventually, rules can fail means that we may need to recover from changes made to control state during the execution of the rule. By not allowing concurrently executing rules, the transaction semantics of rules are simplified.[4]

The actions designated on the right hand side of a rule are executed in sequence until all actions have been successfully completed or until the first failing action. Memory update actions cannot

[4]Transaction and failure semantics for concurrently executing rules is an interesting topic for future research.

fail, but Achieve actions can. In the event of failure, the control state must be recovered.

The sequence of actions on the right hand side of a rule are considered as a single transaction for recovery purposes. If all actions are successful, the rule is complete and execution returns to the package execution module. However, if an action is unsuccessful, any permanent results (i.e., memory updates) of all actions executed by the rule are backed out to the beginning of the rule. This includes removing any alternative queries that were created by intermediate actions.

Execution of an Achieve action depends on the goal of the action. If the goal is represented by a goal package, execution directly transfers control to the indicated package. In this case there is no decision to be made — all decisions about further processing are made in the goal package. If the goal is a primitive goal of the region, execution transfers to the primitive action execution module for that goal.

Primitive Action execution

A primitive action tries to directly satisfy a goal by executing a subordinate region that can satisfy the goal. Since more than one region may be able to achieve a particular goal, a primitive action must choose between the regions. Also, since a region may not succeed at achieving a goal, a primitive action must try alternative regions to ensure that no region can achieve the goal before the action admits failure.

Primitive actions use region applicability information to determine which region to execute to achieve a goal. Static applicability information is first used as a filter to eliminate some regions from consideration. If, after the filter, there is more than one applicable region, the results of dynamic applicability functions executed on the applicable regions are used to further filter out regions or to determine an order for trying similar regions. If the dynamic applicability information cannot distinguish a single region to try, similar regions are ordered randomly. This process is described in Figure 6.13.

A Primitive action sends a copy of the query to be processed to a subordinate region. The subordinate region will make modifications directly to the query copy and, if the region is successful, the primitive Action can update the global query information with the transformed result. The copy semantics put all decisions about maintaining transformed results in the domain of the query manipulation routines. The advantage of this approach is the protection of the parent region's memory. The disadvantage of this approach is that transformed results will usually reference many of the same subqueries as the original query. These references can get lost unless the query manipulation routines can recognize common subexpressions using just the copied query representations.

The search for a region to achieve the required goal requires only a single successful result. The primitive action execution is terminated when a subordinate region is successfully executed, when there are no more regions to try, or when global termination conditions indicate that no more searching is desired.

6.4.5 Handling Failure

The planning system uses the heuristics defined in the planning rules, along with applicability information provided by subordinate regions, to determine sequences of region applications that will transform a query to attain the desired goal. However a region execution will not necessarily achieve the region's goal on every query to which the region is applicable. This results because applicability information provides necessary, but not sufficient, conditions for the query in order for the region to be able to process the query and attain the goal.

Region execution is a primitive action, and region execution failure can propagate into primitive

```
INPUT: query Q
OUTPUT: query, success indicator

    initialize success := false
    initialize continue := true
    use static applicability to filter out inapplicable regions
    if there is more than 1 applicable region, build a priority queue of regions
        (use dynamic applicability then random choice, for ordering)
    while continue ∧ (≪termination conditions≫ not met) do
        allocate termination conditions for subordinate and
        execute first region in queue on Q' ← Q
        if region returns success
            set success := true and continue := false
        otherwise
            remove region from queue
            if queue-empty then set continue := false
    endwhile
    if success then Wm_Apply(Add_Choice, (QuerySet Get_Equiv(Q)), (Q') )
    return Q' and Success?(Q', this primitive goal)
```

Figure 6.13: Primitive Action Execution

action failure, rule failure, goal package failure, and eventually parent region failure. Thus, each of these modules must accommodate the possibility of failure.

The primitive action module handles region execution failure by executing other regions that can achieve the same goal as the failed region. It will try all regions until it finds one that can achieve the goal, or it finds that no region will attain the goal. In the latter case, the primitive action admits failure, which propagates to the rule.

The sequence of actions on the right hand side of a rule are considered as a single transaction for recovery purposes. If any action on the right hand side of a rule fails, the rule itself fails. Any updates made to memory, in particular updates to query choices, must be backed out to the point before the first rule action. This recovery is handled through maintenance of a log of updates to memory state, delimited by transaction boundaries. A transaction, in this case, encompasses the execution of an entire rule.[5] A transaction starts before the first action of a rule is executed and ends when the last rule action is successfully completed. Since rules can be nested within other rules (through Achieve actions) the recovery of a transaction can require backing-out of successful, nested rule executions.

When a goal package terminates without achieving success, that failure propagates to the rule executing the Achieve action that invoked the package. If the goal package is a high-level execution no further recovery is required. In this case the high-level execution module must choose another

[5]There are certainly situations in which one would want to save the results of intermediate actions, since they may offer opportunities for later processing. Potentially good intermediate results could be indicated by incorporating save point actions in rules – where a save point indicates at point at which alternative queries generated by actions of the rule should be made persistent. Since rules are effectively nested, an interesting question is how to handle save points in the resulting nested transactions. We defer an answer to the semantics of save points in nested transactions to later work.

query/goal pair to execute.

Failure of a primitive action or rule will not necessarily propagate to region failure, nor does successful processing of goal packages indicate that the region will be successful at achieving its goal. The success, or failure, of a region to attain its goal depends solely on the definition of success for the goal and the queries that are generated as choices during the region's processing.

6.4.6 Extensibility

A key feature of the planning-based control is that, as a rule system, it is inherently extensible. Rules can be added to existing goal packages to describe new heuristics for combining the optimization processes described by subgoals of the rule system. Goal packages can be added to describe new goals (or subgoals) for query processing.

The addition of a new region to an optimizer may require modifications to the control. A region that works towards the same goal as some existing region provides an alternative way to attain the goal, and does not require any changes to the rule system. The only required modification is that the parent's primitive action for that goal be notified of the existence of the new region.

On the other hand, new regions can reflect completely new techniques for dealing with queries and have goals that are new to the existing optimizer. The addition of such regions to an optimizer would require an understanding of where the goal will fit in relation to other goals of the optimizer. Such an understanding could be pre-existing; i.e. a developer has a particular strategy in mind and develops regions to fit into that strategy. Fitting a new region into an optimizer could also be a process of experimentation. A major advantage of the Epoq extensible control is that it facilitates experimentation with combining optimization techniques to process a query.

6.5 Summary

The Epoq approach to extensible query optimization is embodied by the architecture presented in this chapter. Each region in an Epoq optimizer is a separate module that interacts hierarchically with other modules through a common interface and a planning-based control. The potential interaction of modules is statically defined by control rules, region goals, and applicability, but the actual interaction between regions depends on the query being processed.

A region module provides, through the interface to its parent, a goal for its processing and predicates characterizing the queries it expects to be able to manipulate to achieve the goal. A parent control uses this information as it decides how to process a query.

A parent region must determine an order for executing subordinate regions to transform a query to achieve its own goal. Given a particular query, a region control plans a sequence of transformations (i.e., an ordering of subordinate region executions) that will, hopefully, manipulate the query to achieve the region's goal. The planning process is influenced by intermediate results of the plan, thus planning is interleaved with the execution of subordinate regions.

The Epoq planning-based control was heavily influenced by the OPS5 rule-based programming language [34] and by the reactive action packages of Firby [46]. Our rule execution system though is more directed than either of these – it is very single-minded in its pursuit of tasks. Our rule engine enforces a transaction type of semantics on rules – a rule is a task that (if successful) will result in a desired transformation of a query. Thus we require that a rule execute to completion before any other tasks are considered. A rule describes a consistent way to process a query. Alternative ways, described by other rules, are only tried after first rule has completed. In other words, a rule cannot be interrupted as it pursues its goal.

Planning rules describe heuristics for interactions between regions. These rules also support extensibility in the optimizer. The addition of a new region to an optimizer may require new rules to describe how this region may successfully interact with other regions. These rules are added to the parent's planning system's rule set and manipulated in the same way as existing rules. The extensibility of the control itself is a unique feature of the planning-based control in supporting the extensibility of Epoq.

Chapter 7

Internal Query Representation

The internal representation of a query is critical to the power and extensibility of an optimizer. The query provides an important means of communication between regions; the information that can be stored in the internal representation can be shared by different regions. Thus, it is important that the internal representation be able to store information that an optimizer may want to pass between regions and, since new regions may want to share new kinds of information, it is important that the query representation be extensible. Also, since the purpose of a region is to manipulate a query expression, the representation must express all elements of a query that could be involved in manipulations, and must itself be easily manipulated.

Thus, in order to support optimization in an object-oriented database a query representation must be extensible and must be able to provide support for a variety of query manipulations, including the recognition and manipulation of nested queries. In this chapter we present a query representation that provides this support. Both data and query operators are represented as nodes of a query and, in addition, the representation supports nested queries by treating query predicates at the same level as query operators and data objects. Nodes and edges are also annotated with information that will prove useful in query optimization.

In the next section some alternative query representations are presented, and we discuss why these do not provide the required support for our model. Our tree representation for queries is introduced in Section 7.2. Annotations to this representation are discussed in Section 7.3. Algorithms for building and manipulating trees are presented in Section 7.5 and the query execution represented by a query tree is described in Section 7.4. In Section 7.6 some useful extensions of the tree representation to a graph are presented, and the implications of these extensions are discussed.

7.1 Some Background

Representations for algebraic expressions can be categorized as either operator-based or class-based, where class refers to a set of data items of the same type. An operator-based representation is an operation tree where internal nodes are operators, leaves represent input data, and edges indicate an ordering (usually, bottom-up) of the operations. Class-based representations are trees (or graphs) in which nodes represent data and edges represent operations. Here, operations are viewed as connections between data items. For example, class-based representations are often used to represent join operations; an edge between classes represents a join between those classes.

7.1.1 Class-based representations

Class-based representations are applicable to data models in which each data type has an associated collection of instances of all data items having that type (i.e., an extent). In such systems a query can be viewed as a graph of relationships between the extents. Such relationships may actually be operations on the datasets. For example, class-based representations have been used in relational query processing for determining join orderings (e.g., [18]). In this case a relation plays the role of a class (the relation scheme is the type and the tuples form the extent), and edges between class nodes represent join predicates. This type of representation proved useful in processing distributed queries using semijoins [17].

A class-based representation is used for processing distributed queries in the Orion database system [75]. In this system, each type has an associated class, and attribute connections between types imply connections between the classes. For example, if type Employee has an attribute **Dept** of type Department, there is a connection between the classes Employee and Department. Similarly, such connections in a query expression (e.g., x.Dept) result in connections between classes in the query graph representation. The model has been applied thus far only to queries that Select over a single class, so the only operations represented in the queries are relational comparisons of attributes (e.g., x.Dept.budget $>$ 1000000) and Boolean connections between them (e.g., x.Dept.budget $>$ 1000000 AND x.Manager.salary $<$ 50000). The class extension graph is primarily used to determine execution orderings for traversing path expressions, distributed site selection for subquery executions, and potential parallelism in processing branches of the query tree.

A representation for queries proposed by Cluet and Delobel [31] unifies algebraic rewrite, class extension optimization, and factorization of common subexpressions through a class-based representation described operationally by an algebraic expression. The representation supports Select-Project-Join queries (also Union and Intersection) where both the Select and Join involve simple predicates over class attributes. The representation includes a graph (not necessarily connected) representing class connections in the query, a tree representing a selection predicate for the query, and another tree representing a join predicate.

The class-based representation can be manipulated based on heuristics related to physical storage. Although type extents are represented in the graph, those extents may not actually exist in the database (they are called *virtual extents* by the authors), and manipulations over the class-based representation consider such situations. The algebraic representation of the query can be manipulated by algebraic transformations.

Lanzelotte et al. [91] use a class-based representation specifically to support optimizations for path traversals expressed in a query. Path expressions represent an implied traversal between the classes involved in the expression. For example, the path expression a.dept.manager represents a traversal from the class of Employees (assuming a is an Employee) to the set of Departments to the set of Managers. The steps of such a traversal are called *implicit joins* by the authors. The potential for such a traversal is represented by the attributes of the different types in the database (e.g., Employees has an attribute **dept**), and a class-connection graph combines the statement of such traversals with explicit join operations in a query. The edges in the class-based graph represent implicit joins (for traversals) or explicit joins (i.e., join operations between classes not connected in the schema). Selection predicates on single relations are also represented by a cyclic edge on the node representing the relation.

The class-based representation is used to construct an operator-based tree involving Select and Project operators and three kinds of Joins (implicit, explicit, and indexed). The operator-based tree can then be transformed according to rules involving the operators in the tree.

7.1.2 Operator-based representations

The representation of a query by an operator tree (or graph) is useful for manipulating the query expression since the representation is tied very closely to the algebraic query expression. Nodes in an operator tree represent query operators (a logical level of operation) or access methods (physical level of operation). A query represented as an operator tree can also be translated into an executable plan for query execution. For example, Smith and Chang [138] use transformations over an operator tree to optimize an algebraic expression, then translate that tree to determine an efficient execution plan for the expression. Operator trees manipulations have also been used to determine orderings for join methods (e.g., [87] and [91]).

Graefe [54] uses operator trees to support extensibility in queries over complex objects. Since the sets of operators, access methods, and transformations are independent of the representation, any extensions are automatically supported by the representation. A new operator or method simply becomes a new label for a node in an operator tree. That label may be recognized by new transformation rules.

In Graefe's representation child nodes of a query operation or access method represent computations that provide input data to that function. Input to an operation can be the result of another query (i.e., a nested query). Operator nodes usually include any predicates associated with the operator (e.g., a Select predicate) and these predicates can be manipulated by code provided in transformation rules. Query transformation rules describe manipulations over an operator tree that include re-ordering of query operators and translating query operations to access methods affecting those operations.

In the Starburst system [64] extensibility is supported by their *Query Graph Model* (QGM). A QGM is essentially an operator graph, although it has to some extent the flavor of a class-based graph. In the model, high-level operations such as Select are represented by boxes and lower-level operations (such as set iterators) are represented by vertices. These vertices are enclosed in the boxes; the vertices interpret the access to data that is required to satisfy the high-level operation. Edges connect vertices with other vertices (in the same box or other boxes) and may also connect vertices with boxes. In the latter case the boxes represent data which is input to the operation indicated by the connected vertex. Transformations can be written to reorganize any of the components.

This representation supports nested queries through the vertex-box connections and is extensible by adding new definitions for any of the components. For example, a new type of Selection would require a new definition for a box, which would include an interpretation of the query expression into vertices and edges for that box. The vertices might also represent new operation types which would have to be defined. New transformation rules could also be added to the system to manipulate queries involving the new definitions.

Rosenthal and Helman [121] argue that operator graphs are a requirement for supporting extensible optimization and propose an operator graph approach that includes nodes to represent data that is output from, as well as input to, an operation. These data nodes are labelled with predicates describing the data they represent. For example, a data node might be labelled " R1 restricted by name > 'S'" or "R1 Join R2".

The authors point out that data nodes provide not only a central location for information about the data (such as predicates, dataset size, best access cost) but also allow the representation of alternative plans for computing a subquery, i.e., multiple children of a data node represent different ways the same data can be computed. They also note that data nodes provide a single place to add unary operations (such as Sort) to an operation sequence, a single place to represent common intermediate results when optimizing multiple queries, and a way to represent "convenient"

intermediate results (such as cheapest relation regardless of sort order).

Transformations defined over the graphs must also contain information for updating predicates and other labels for data nodes. Extensibility in query processing is supported, in this representation, by extending the set of data node properties and extending the collection of graph transformations. The latter supports new transformations involving new operations and methods, the former supports any new labels these transformations may want to use on data nodes.

7.1.3 Comments

None of these representations is completely satisfactory for representing queries over an object-oriented data model. The extensibility of the data model means that a query representation must be easily extendable to handle new data types, operations, access methods, and transformations. Operator-based representations can easily support such extensibility, but class-based representations derive much of their meaning from fixed relationships and operations, and are thus not very extensible. In particular, class-based representations emphasize join-like operations and it is not at all clear how they might be used to represent arbitrary algebraic operations as may be found in an object-oriented database. Even when representing join operations, class-based representations require modification to the representation when new kinds of joins need to be considered [38].

Operator trees have been shown useful for supporting the algebraic rewrite of query expressions [121], while class-based graphs are useful for manipulating path expressions [75, 91].

A major shortcoming of these representations is the lack of support for subqueries. Class-based approaches do not support subqueries at all, although Cluet and Delobel note that they are working on that problem [30]. Operator-based approaches can support queries nested as input to another query but do not support query expressions in the terms of predicates. This results because these representations consider operations and data as the primary objects in the representation, and subjugate predicates involving the data to adornments of the nodes or arcs [54, 64, 91, 121] or to separate structures which complement the primary representation [31].

In an optimizer for an object-oriented system, however, it is important that predicates be treated at the same level as query operators and data. Many object-oriented languages allow the expression of subqueries in any part of a query. For example, in O_2Query a subquery can appear in the Select, From or Where clauses [110]. An object-oriented query could also involve methods that are themselves implemented as queries.[1] Such methods could appear at any point in a query. For example, if the **avgsal** attribute of a Department is implemented as a query, the query "Select tuple[D:d, Sal:d.avgsal] From d in Departments Where p(d)" involves a subquery in the Select clause. If **avgsal** is implemented as a query that retrieves the salary of all employees of a department and computes the average, we would want to be able to optimize the employee retrieval. In order to allow the optimizer to work on all parts of a query, it is important that the internal representation be able to express nested subqueries in a way that allows them to be detected and manipulated by the optimizer regions.

7.2 A Tree Representation for Queries

A query is represented as an extensible annotated tree (an EAT). A query tree is composed of data nodes and function nodes, connected by labelled, undirected arcs.[2] This representation generalizes operator tree representations (e.g., [54], [121]) by treating algebraic operations, other methods

[1] This isn't unique to object-oriented models. A similar problem could result in queries involving type Postquel in Postgres [125], for example.

[2] In Section 7.6 we discuss extending this representation to a graph with directed arcs.

defined over objects, and arbitrary functions over objects (e.g., predicates and function parameters) uniformly.

Data nodes represent data that is manipulated as part of executing the query. A Data node can represent an object in the database or can represent an object built by the query, a subquery or some other function. *Function nodes* represent actions that can be taken on data. Thus, a Function node will always have at least one child Data node, representing the input to the function, and one parent Data node, representing the output of the function. Additional children represent the data provided by other parameters to the function.

Nodes are (optionally) labelled with syntactic query information. Function nodes will be labelled with function names. A Data node may be labelled with a name for the data object represented by the node. For example, in Figure 7.1 the leaves are labelled with the names 'People' and 'p', respectively.

Nodes may also be annotated with other information about the query. For example, type information in Figure 7.1 is provided as an annotation to the Data nodes. The annotations available on a node are defined by a database implementor, and are intended to provide more information than query syntax. This is discussed more fully in Section 7.2.1.

Nodes are connected by *arcs* representing relationships between data and functions in a query. An arc can only connect a data node with a function node; one of which is designated as the parent and the other as the child node of the arc. Although these arcs are undirected, for a given node A we usually refer to an arc connecting A to a child as an *incoming arc* for A and an arc connecting A to its parent as an *outgoing arc* of A.

The type of an arc is determined by the parent type. An arc connecting a Data node, as parent, to a child Function node is called a *DF_Arc* and an arc connecting a parent Function node with a child Data node is an *FD_Arc*.

A DF_Arc represents the fact that the data (i.e., parent node) is produced by the execution of the child function. DF_Arcs are not labelled. As will be seen in Section 7.6, a data node could have more than one incoming DF_Arc, representing the fact that the same data can be produced by different functions.

An FD_Arc represents the fact that data is supplied to a function. We distinguish between FD_Arcs that relate a function to its input data and FD_Arcs relating the function to its other arguments. Input data is the data upon which the function operates; other parameters to a function are functions and predicates that are applied to the input data. An FD_Arc connecting a function node N, representing a function f, to a node representing an input dataset for f is called an *input arc for N*. When there is more than one input arc for a function node, these arcs are ordered. This ordering supports functions that distinguish between their input datasets by position — e.g., Join access methods.

FD_Arcs may be labelled. These labels can be used to associate names with function parameters. For example, the EQUAL query *Ojoin(Set1,Set2,A,B,λa,b p(a,b))* associates variable *a* and name *A* with members of input *Set1* and variable *b* and name *B* with members of *Set2*. Variable names are used to associate members of the input to manipulations in the predicate p(a,b). The field names (A and B) are used as field labels (i.e., attribute names) in the tuples output from the Ojoin.

Any type of arc may be annotated. In general, arc annotations are used to capture information about the execution of a query. For example, in Section 7.3 arc annotations are used to capture information about the availability and usage of variables in subqueries.

The root node of a tree is a Data node representing the output of the query represented by the tree. It has one incoming DF_Arc, connecting it to a child function node representing the query operation computing the result. The root node has no outgoing FD_Arc, although it is sometimes useful to represent the root node as the child of an arc with no parent. This arc, called the *root arc*,

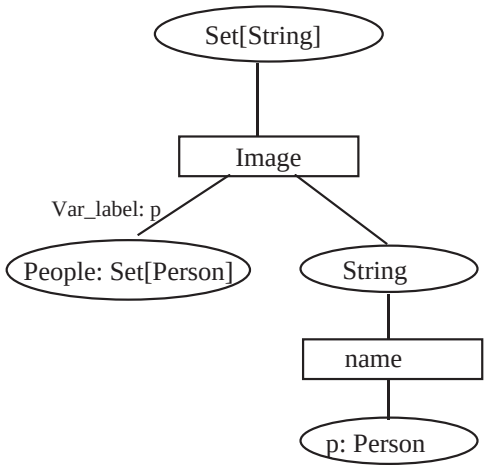

Figure 7.1: A tree representation of Image(People, λp p.name).

is a convenience to allow us to consider data (constants) passed to a query embedded in a program. As will be discussed later, an arc can be annotated with information (e.g., variable availability) that is accessible to subtrees below the arc. We do not show root arcs in any of the examples here.

As an example, consider the simple query *Image(People, λp p.name)* which creates a list of names. Assume the People object has type Set[Person], and the **name** method on type Person returns a string. This query is represented as in Figure 7.1, where Function nodes are rectangles and Data nodes are ovals. In this example, each Function node is labelled with the name of the function it represents and each leaf Data node is labelled with a constant or variable name. Each Data node is annotated with the type of the data represented by the node. The root node is a Data node representing the query output; a set of strings. It is connected to a Function node labelled Image which represents the Image function that produces the query output. The Image function has two parameters, thus the Image node is the parent of two Data nodes. The left child represents the input set — i.e., the People set object. The arc from this child is labelled with variable p to indicate that p is the name associated with an input set member.[3] The right child of Image represents the result of applying the **name** method to object p — i.e., the result of the lambda function defined by Image applied to an element of People. This child is the root of a subtree representing the application of function **name** to an object called p.

7.2.1 Annotating Nodes and Arcs

All tree components (nodes and arcs) can be annotated with information about the component, and about the data, function, or relationship embodied by the component. These annotations differ from labels, which convey syntactic information about a query expression and are therefore part of the basic query representation. Annotations are used to store information about a query and the objects it manipulates that may be useful in the optimization process. For example, annotations can be used to indicate the type of a data node. This type information could be used, for example, in type inference [147] or in semantic query optimizations [101].

Annotations are a way in which more information than the query syntax can be passed between

[3] In subsequent figures we only indicate Var_labels for join operations, where it is necessary to distinguish between variable names assigned for two input datasets.

regions. For example, one expected use of annotations is to support a cost model for queries. An annotation on a data node could, for example, indicate the best cost known for building that data. On the other hand, annotations could carry information that would be used by an optimizer to calculate cost. For example, the cost model of Bertino and Foscoli [21] requires information such as type, the number of members of a set or the size of the range of a function (method) as parameters to cost computations. This information can be easily stored as annotations to nodes or arcs.

An *annotation template* describes the annotations that can be attached to a component of the representation. There is one template for each kind of component in a representation; i.e., one Data node template, one Function node template, and one Arc template.

DEFINITION 7.1. *An annotation template is a table of (Name, Type) pairs where*

- *Name is a value that can be used as an index.*
- *Type is an abstract data type name.*

All Names in one template will be values of the same type; for example, all Names in a template will usually be Strings. Also, in a single template all Names are unique, since this field is used as an index. The template associates a Type with each Name, indicating the type of data that can be associated with that Name in an instance of the template.

An *annotation table* is an instance of an annotation template and, as such, is collection of annotations – at most one annotation for each (Name, Type) pair in the template. Basically, the annotation table fills in the template.

DEFINITION 7.2. *An annotation (in table T) is a pair (Name, DataValue) where*

- *Name is a Name in the template of T.*
- *DataValue is an object of the Type associated with Name (in the template for T).*

For example, suppose all data nodes need to be annotated with type, size and representation information. Then the Data node template specifies a table of three elements, one with name **Type** and type String, one with name **Size** and type Real, and the other with name **Rep** and type String.[4] A query node representing a database set of 200,000 Person objects stored as an Ordered_List could have the following annotation table:

Name	DataValue
Type	Person
Size	2.0E05
Rep	Ordered_List

The collection of annotations that can be associated with nodes or arcs is determined by the optimizer implementor by defining an annotation template, and can be extended as needed by adding to a template. For example, the addition of a new region could require new annotations that will be used by that region in its optimization process. A region only uses annotations that it recognizes, so the additional annotations will be ignored by previously existing regions.

[4] We represent types and representations by their names for ease of reading.

7.2.2 Subtrees

A primitive subtree of a query consists of a function node along with its parent (output) data node, input and parameter child data nodes, and corresponding arcs, labels, and annotations. Figure 7.2 shows primitive subtrees corresponding to some of the EQUAL operators. In the figure, function nodes are labelled with the function name and data nodes are annotated with type information.

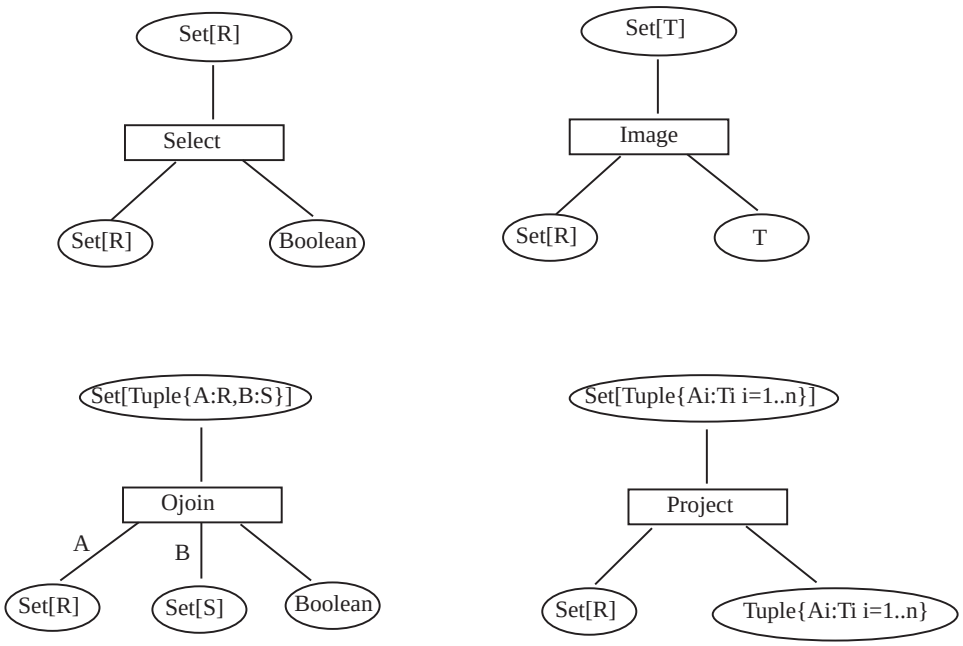

Figure 7.2: EQUAL Primitives

Since all functions have at least one input and one output, all leaves of an EAT are data nodes and the root node of an EAT is a data node. Function nodes must be internal nodes.

The leaves of a primitive subtree may themselves be roots of a query tree. For example, in Figure 7.2 the Project subtree is simplified by assuming the existence of a separate tuple-building operation whose output is represented by the rightmost leaf node (annotated with type $Tuple\{A_i : T_i \; i = 1 \ldots n\}$). Similarly, the rightmost leaves of the Select and Ojoin operations represent the results of the execution of Boolean predicates over the inputs of the Select and Ojoin, respectively, and the rightmost leaf of Image represents the result of executing a function over members of the Image input.

Leaf Nodes

Leaf nodes can represent either constants or variables. Constants are objects that are defined somewhere outside the scope of the query. Constants will usually be database objects or database constants (i.e., primitive objects such as numbers or strings). For queries embedded in programs, constants can represent program data objects. Nodes representing constants may be labelled with a name for the constant. In Figure 7.1, for example, the node labelled People represents the database set with the name People. This set is a constant in the query.

Variables represent objects that will be bound when the query executes. They are named within the query and usually reference members of a constant set of objects or a set of objects built by a

subquery. For example, in Figure 7.1 the node labelled p is a variable node; the variable represents a member of the People set and p is the name assigned in the lambda function of the Image operation.

7.2.3 Comparisons

The extensible annotated tree generalizes operator trees in two basic ways. First, all parameters of a query operator are treated as first class objects and represented in the tree. In operator tree representations, only the input and output datasets of a query operator are represented; parameters such as Select predicates are treated as part of the operation that uses them. For example, a standard operator tree representation of the query of Figure 7.1 (Image(People, λp p.name)) would contain a data node for People, a data node representing the Image output, and an operator node for Image. Any information about the function application p.name would be captured with the Image operator node.

The fact that all parameters are first class in the EAT representation means nested queries are more fully supported by EATs than by operator trees. In particular, queries nested in predicates or function parameters, as well as nested input queries, are expanded in an EAT. This allows for optimization strategies that can manipulate subqueries nested in any part of a query. For example, rule-based optimizers that search for matches between rules and subqueries will explore subqueries nested in predicates, since they can explore anything represented as trees. In addition, specialized strategies (such as the CJ region in Chapter 8) may be able to manipulate the expanded predicates.

A second way in which an EAT generalizes operator trees is in the extensibility of the annotations. Providing annotations as an extensible set of properties, rather than as fixed characteristics of a node or arc, allows an implementor to define the kinds of information that need to be available about the components of a query. This supports the addition of new strategies that may need new information about a query.

Annotations also provide a means for supporting a cost model in an optimizer. Information about query cost, as well as data for computing query cost, can be stored in annotations and maintained through methods that manipulate the annotation values.

7.3 Annotations for Nested Queries

As noted earlier, the kinds of annotations associated with nodes or arcs are determined by a database implementor and depend on the kinds of information needed for query optimization. In this section, we discuss annotations that provide information about the scope and usage of variables referenced by a query. These annotations support the expression of nested queries. In particular, they can be used to support queries nested in predicates, as in EQUAL.

NOTATION. Any annotation X for a given node or arc N is denoted as X_N.

Query operations may define new variables that can be used by some of the subqueries of the operation. For example, the EQUAL query $Select(A, \lambda a\ p(a))$ contains a lambda function defining variable a for the predicate subquery $p(a)$. To capture this information we define an annotation named *Def* with type Set[Variable]. For node N representing query operation f, the *Def set of N* (Def_N), also called the *Def set of f*, is the set of variables defined for operation f. A Def set only makes sense for a function node, so the following definitions hold:

$\text{Def}_N = \{\ \}$ for N a data node
$\text{Def}_N = \{\text{variable}\}$ for N a function node

The Def set of a function node is populated by variables defined for parameters of the function represented by the node. For example, in the EQUAL query

Select(A,λa p(a))

the Def set for the Select function node is {a}. For the SQL-style query:

Select	f(x)
From	x in X, y in Y
Where	p(x,y)

the Def set for the Select function is {x,y}. In EQUAL, the Def set for an Ojoin will contain two variables, the Def sets for Image, Select and Project will each contain one variable, and the Def set for any other operators will be empty.

Def sets do not always provide enough information to bind variables to objects for query execution. In particular, for multiple input queries, information about bindings of input set members to variables is required. This information could be stored by defining the Def set as an ordered list; with the variables in Def ordered with the same ordering as the input arcs to the operation. Rather than overload Defs in this way, however, we choose to label input arcs with variable names as described in Section 7.2.

Arcs can be annotated with information about the availability and use of data variables in the query.[5] The annotation template for an arc includes two items, both of type Set:

Use:	Set[variable]
Avail:	Set[variable]

Avail is the set of variables available for use by a Data or Function node; it corresponds to a symbol table for the child node of the arc. The Avail variables are associated with the Parent to Child direction of an Arc; the variables available to a node are also available to its children. Note that this set allows us to represent variable availability in nested subqueries. A child node of a function could be the root of a subtree that represents a query nested within that function. The Avail set is passed to all children in the subtree and will represent variables that can be referenced by any functions in that subtree. A Function can only reference variables in the Avail set of the arc connecting to its parent.

Use is the set of variables actually used by a subquery. The Use set on an arc lists the variables actually referenced by the descendants of that arc. Thus, the Use set details variable usage in nested subqueries.

For example, consider Figure 7.3 which is an annotated representation of the EQUAL query Image(People, λp p.name). The variable p is considered to be defined by the Image function, so is in $\mathrm{Def}_{\mathrm{Image}}$. This variable is available for use by all components of the subquery representing the application of p.name, thus is in the Avail sets of the arcs on the right subtree descending from Image. Since the variable p is referenced at the leaf of this path, p is also in the Use sets for the arcs on that path.

Consider Use and Avail from the viewpoint of the child of an arc. Avail contains the variables that can be used in expressions represented in the subtree rooted at that child. Use contains the variables actually referenced in expressions in that subtree. If Use contains a variable not in Avail, there is a syntax error in the subquery. This can be represented as the following axiom:

AXIOM 7.1. *For any arc A,* $Use_A \subseteq Avail_A$.

[5] This information is attached to arcs because it is related to the flow of data in a query. The execution model for a query defines that data flows along the arcs.

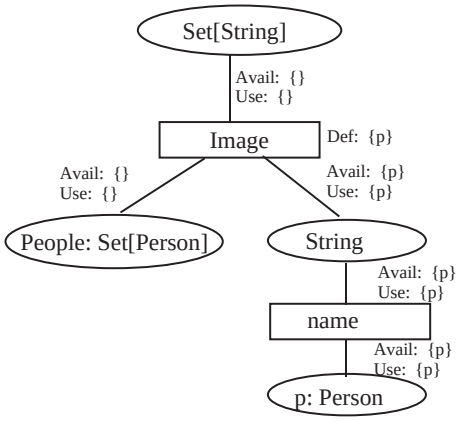

Figure 7.3: An annotated representation of Figure 7.1. Annotation tables for arcs and function nodes are shown next to the component. Type annotations are shown inside the data node.

In EQUAL, as in most query languages, the scope of variables defined at some function node N representing function f includes all parameters to f except the input parameters. For example, in Figure 7.3 the scope of variable p does not include the People input set. This observation can be represented, for EQUAL, as an axiom:

AXIOM(EQUAL) 7.2. *Given node N, parent arc P and child arcs P_i, i=0...n such that $\forall$ i, P_i is not an input arc for N,*

$$Avail_{P_i} = Avail_P \cup Def_N$$

In EQUAL, we do not allow variable names to be overridden in nested subqueries. This can be expressed as follows:

AXIOM(EQUAL) 7.3. *Given node N and arc P connecting N to its parent*

$$Avail_P \cap Def_N = \phi$$

7.4 Execution Model

The fact that all operator parameters are treated uniformly means that the execution represented by an EAT depends on the execution semantics of each of the query operations represented in the tree. Similarly, this fact, and the presence of annotations involving data type and variable usage, means that the correctness of an EAT depends on the syntax and semantics of the query operations and all parameters to those operations.

Unlike an operator tree, in which the evaluation is implicitly bottom-up, the query operation represented by an EAT is more correctly described as a top-down recursive execution. This results from the fact that subtrees representing non-input arguments to a function usually require data provided by the input arguments to that function. Input arguments flow up the tree to the operation consuming them, and are processed by moving down to the subqueries representing the non-input arguments to the operation. For example, in the query $Select(People, \lambda r\ r.age > 15)$ (subquery at node 16 in Figure 7.4), the subtree representing the parameter function $r.age > 15$ is executed once for each r in the People input set. In other words, each object in People moves up the tree

to the Select function where it is bound to the variable r and passed down the tree to the Boolean predicate. The Boolean result is, for each r, passed back up the tree to the Select function, where the Select result is collected.

The Use annotation for arcs provides further information about the actual operation of a query, since Use captures the actual objects (variables) that are required for a subtree to execute. A subtree might require no variables, variables that are instantiated by its closest ancestor operation (i.e., the operation within which it is immediately nested), or variables that are provided from a more distant ancestor. For example, the Boolean operation represented in node 22 of Figure 7.4 uses the p1 variable defined by its immediate function ancestor (the Select of node 15) as well as the o variable defined by the Image function represented by node 8. The Vehicles set, represented in data node 3 of the figure, has an empty Use set since it is constant relative to the query.

The Use annotation of an arc details the variable usage of the subtree rooted at the child of the arc. We say that that child uses (or requires) the variables denoted in that Use set. In other words, a node requires the variables denoted in the Use set of its outgoing arc. As noted in axiom 7.1, a node can only require variables defined by some ancestor of the node in the tree.

We call a subtree that requires variables defined at ancestor F in order to execute a *dependent subtree of F* or *dependent subquery of F*. Conversely, any subtree that can execute independently of F is an *independent subquery of F*. In other words,

DEFINITION 7.3. *Given data node D, parent arc P of D, and any ancestor node F of D, the subtree rooted at D represents a dependent subquery of F if* $Use_P \cap Def_F \neq \phi$. *Conversely, the subtree rooted at D represents an independent subquery of F if* $Use_P \cap Def_F = \phi$.

Note that a subtree can be dependent and independent at the same time. For example, consider the subtree rooted at node 14 in Figure 7.4. This subtree is dependent on node 8 (i.e., the Image function) because it uses the variable *o* defined at that node. However, the subtree is Independent of any ancestor node numbered smaller than 8, because the subtree does not use variable v.

Subtree independence can be useful in determining query execution order. For example, consider the subquery rooted at node 16 in the example. This subquery has an empty Use set, indicating the subquery is independent of all ancestor query nodes. Thus, the value computed by the subquery is constant relative to the rest of the query. As a result, the query $Select(People, \lambda r\ r.age > 15)$ could be executed once and saved for use when needed in the execution of the remainder of the query.

7.5 Tree Manipulations

An annotated query tree is similar to an annotated syntax tree for a query, and building an annotated tree is similar to syntactic and semantic analysis in a compiler [6]. The syntax of the query is stored in function nodes and, to some extent, in arc labels. Annotations store semantic kinds of information – data types, cost information, variable usage information, etc.

A region transforms a query by manipulating a tree. These manipulations can involve rearranging or replacing subtrees, or changing annotations on nodes or arcs. In general, any modification to a node or subtree can require modifications to the annotations of all nodes and arcs in any path to that subtree. Such modifications can be done as an updating pass over the entire tree to recompute annotations, or might be done incrementally. We assume for now that such modifications require an updating pass over a query tree, and leave the problem of incremental modifications for future research.

We can think of building a tree in two phases. In the first phase a query expression is parsed and the tree is built. In the second phase, values for annotations are computed and added to

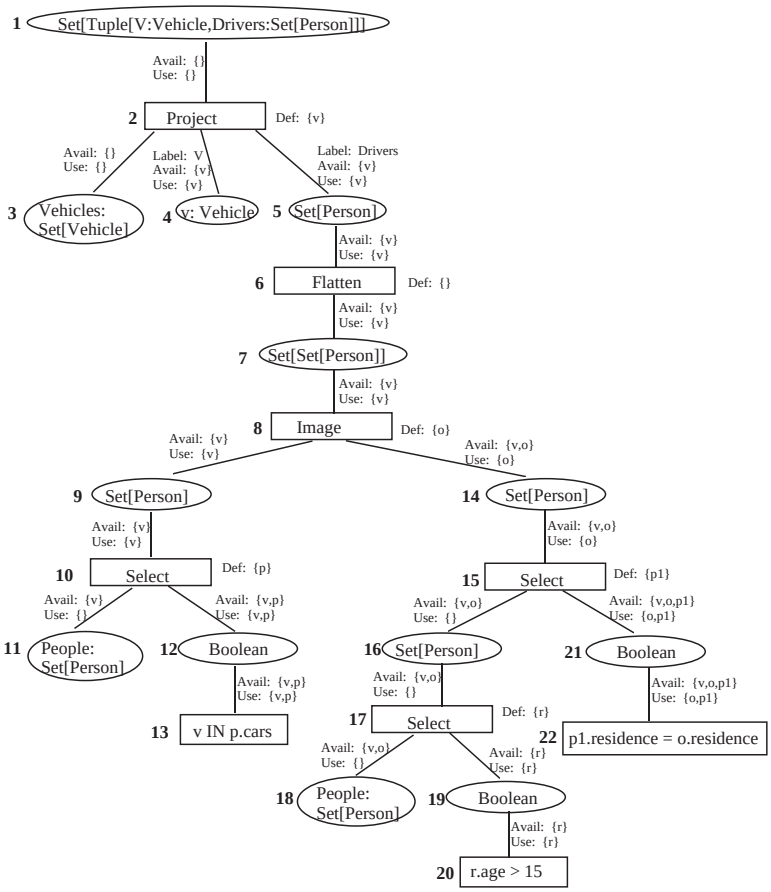

Figure 7.4: An annotated query tree

the tree. In practice, these phases do not have to be separate. Combining them would be more efficient for tree-building. On the other hand, updating a query tree could reuse the second phase of tree-building, since changes to the operations of a query tree could result in (non-local) changes to annotations. Also, a separate phase for annotating provides better support for extensibility, since all new annotations are handled in one place.

In a query tree, some annotations are inherited from parent node and arc information, and some annotations are synthesized from child node and arc information. Inherited annotations correspond to annotations that can be computed while traversing down a tree, while synthesized annotations can be computed while returning back up a tree. Of the annotations previously discussed, Avail and Def are inherited, and Use and Type information are synthesized. Labels are inherited; they correspond to syntactic information available at the node being labelled or at the parent node of an arc being labelled.

The algorithms given in Figure 7.5 describe building an EAT from a query string, and annotating with Type, Def, Avail and Use information. The tree-building process is modularized as a process that builds a subquery rooted at a data node and one that builds a subtree rooted at a function node. The two processes alternate to build the query tree. The main structure of the tree is determined by the Build_Function procedure. This is because Data nodes have a single child, and the children of a Function node are described by the function parameters. Function parameters

Build_Data_Subtree

```
Input:   query Q, FD_Arc A
Output: An annotated subtree representing query Q, connected to A

Process: D := New_Data_Node
         If Q is primitive then /* initialize labels and annotations */
              label D with the name for Q,
              annotate (D, Type := type of Q),
              if Q is a leaf representing a variable x then
                annotate (D, Use := {x})
                else annotate(D, Use := {})
         Otherwise
              create a new DF_Arc called A',
              annotate (A', Avail := Avail_A) /* update inherited annotations */
              call Build_Function( Q, A', return_type)
              annotate (D , Type := return_type) /* update synthesized annotations */
              annotate(A, Use := Use_A')
```

Build_Function

```
Input:   query Q, DF_Arc A
Output: Data type of Q, a subtree representing function Q connected to arc A

Process: Parse Q by breaking it up into operator name and parameters
         F := New_Function_Node
         annotate(A, Use := {}) /* initialize annotations */
         Label F with the name of the function represented by Q
         For each input dataset parameter I of Q
              Create a new FD_Arc called A',
              annotate(A', Avail := Avail_A), /* update inherited annotations */
              label A' (as necessary),
              then call Build_Data_Subtree(I, A')
              annotate (A, Use := Use_A∪ Use_A') /* update synthesized annotations */
         For other parameters P of Q, in order,
              If P has lambda variables, add them to Def_F
              Create a new FD_Arc called A',
              annotate(A', Avail := Avail_A∪ Def_F ) /* update inherited annotations */
              Call Build_Data_Subtree(P, A')
              annotate(A, Use := Use_A∪ Use_A' − Def_F /* update synthesized annotations */
         Attach F, as the child, to DF_Arc A
         Return the result type of F
```

Figure 7.5: Building an annotated query tree

are all handled similarly; an arc is acquired to connect a Function node to a parameter, inherited annotations (Avail) are determined, and the Data subtree is built. Queries with arguments that are lambda functions contribute variables to the Def set of the Function node, and arcs to these arguments are annotated to include the new variables in their Avail sets.

The parent data node of a Function is annotated with a type that is determined by the nature of the query operation. This type is computed, after the function node is built, from information about the function and information about the types of the children of the function node.

The check for a primitive query (i.e., constant or variable) in Build_Data_Subtree ends the mutually recursive tree building by creating a leaf for the tree. The leaf is initialized with labels and annotations.

The tree building is started by sending the query string (Q) and an annotated FD_Arc (i.e., a root arc) to the Build_Data_Subtree procedure. In general, we would expect the FD_Arc to be already annotated with an empty Avail set. However, a query tree could be built within a program that needs to pass program variables to the query. These program variables are treated as constants in the query and can be passed to the query in the root arc Avail set.

As an example, consider again the insurance query from Chapter 5 (page 69). We repeat the query here for completeness:

> For each vehicle, list all possible drivers. Anyone who is over the age of 15 and lives in the same house as an owner of a car is a possible driver of that vehicle.

Expressed algebraically,

```
Project(Vehicles,λ v [V:v, Drivers:
        Flatten(Image(Select(People,λ p v ∈ p.cars),
                λ o Select(Select(People,λ r r.age > 15
                        λ p p.residence = o.residence)))
                ])
```

The query specifies a result type of **Set[Tuple[V:Vehicle,Drivers:Set[Person]]]**.

To solve the query the Project operation creates a set of possible drivers for each vehicle. This is done by creating a set of all people over the age of 15 (i.e., *Adults := Select(People, λr r.age > 15)*) and, for each vehicle, a set of owners of the vehicle (i.e., *Owners := Select(People, λp v ∈ p.cars)*). The Image operation computes, for each person in the set of owners, a set of co-residents by selecting from the set of adults those people who have the same residence as an owner. The result of the Image is a set of sets, requiring Flatten.

This query illustrates a variety of situations that arise in the representation of object-oriented queries. The query is quite deeply nested so there are a number of different variables with nested scopes, as well as database objects and constants. Subqueries are nested as input arguments and as predicates. There are also dependent and independent subqueries, and there are a number of path expressions in the query.

The EAT of Figure 7.4 shows the result of building an annotated tree for this query. The tree is built using the algorithms for Build_Data_Subtree and Build_Function in Figure 7.5. The nodes of the EAT are numbered to aid in tracing the operation of the algorithms. The nodes are built in the order numbered and arcs are built immediately before their child node. We will not actually trace the building of the tree, but will discuss some of the actions that are taken in the alternating calls to Build_Data_Subtree and Build_Function.

Nodes and arcs are built, and node labels (function names and constant names), Def and Avail set values are computed, as the algorithms move down the tree (i.e., inherited). For example, the

labels on the arcs from nodes 2 to 4 and 2 to 5 are computed by the Build_Function procedure as it builds node 2, before it calls Build_Data_Subtree to build nodes 3, 4 and 5. Data type and Use set annotations are computed as the procedures return — i.e., move back up the tree (synthesized).

At each function node, the query is parsed to find the arguments for the function. For example, the Image query is parsed into three pieces: 1) the operator name (Image), 2) the input (Select(People, λp v $\in$ p.cars)), and 3) the lambda function (λo Select(Select(People, λr r.age $>$ 15), λp p.residence = o.residence)). The query operator is placed in a new function node (node 8), and arcs are built to attach that node to subtrees representing the other parameters.

The Def set for a function node is computed from any lambda function argument to that function. For example, the Def set for the Image function at node 8 contains variable *o* from the lambda function argument of Image. The Avail set for the FD_Arc to a child representing function input is the same as the Avail set supplied on the parent arc of the function node. For example, the arc from 8 to 9 has the same Avail set as the arc from 7 to 8. The Avail set for the FD_Arc to a child representing a lambda function additionally contains the lambda variable from the Def set of its parent function. For example, the Avail annotation on the arc from node 8 to 14 is computed as the union of the Avail set from 7 to 8 and the Def set of node 8.

Procedure Build_Data_Subtree is called on each of the function arguments to build subtrees representing those arguments. Each interior data node (e.g., nodes 9 and 14) is used as a placeholder until the algorithms move back up the tree and annotate the nodes with type information. Leaf data nodes can be simultaneously labelled and annotated (e.g., nodes 11 and 18).

Note that there are some abbreviated subtrees in the example. All predicates have been represented as single function nodes, although a predicate would normally be expanded into a subtree. Path expressions are also not expanded in the figure, but are represented as part of a predicate node. The identity function computing the V attribute of the Project operation has also been abbreviated and is represented only as a single data node (v: Vehicle). These abbreviations are made to save space and help alleviate the cluttered nature of the Figure, and are not made in the actual representation.

Also note that Project is represented as a query operation with a variable number of parameters. In this example, each of the tuple fields is considered to be a parameter of the Project itself, rather than a parameter of an additional tuple-building function that is an argument to Project. This choice abbreviates the representation itself and has no effect on the power or correctness of the representation. Of course, the choice made must be clear to all regions using the representation. In particular, regions which express rules as tree manipulations need to know how Project operations are represented as trees.

7.6 From Tree to Graph

Thus far, we have been representing a query as a tree. The root node and all leaf nodes are data nodes and, in any path in the tree, data and function nodes alternate. This representation can be extended in a variety of directions to represent more information that may be used in transforming a query. The tree representation can be extended to use directed arcs, to allow multiple children of a data node, and to allow a node to have more than one parent (i.e., a singly-rooted graph). We discuss the implications of such extensions in this section.

7.6.1 Directed Arcs

The tree representation can be extended by replacing each undirected arc in the tree with directed arcs, where the arc directions represent the flow of data during the execution of a query. Thus,

an undirected arc can be replaced with one or two directed arcs with data-to-function direction or function-to-data direction, as appropriate. For example, consider the arc connecting nodes 8 and 9 in the example query of Figure 7.4. This can be replaced with two arcs. A single Vehicle object flows in the function-to-data direction (8 to 9) each time the Image function is executed. This object is used in the predicate of the nested subquery. The data-to-function arc (9 to 8) represents the data provided by the input data set (represented by node 9) to the Image function (node 8).

Although nodes would be connected in both directions in many nested query situations, there are occasions where two nodes would only be connected in a single direction. For example, consider the arc connecting nodes 15 and 16 in the example of Figure 7.4. The empty Use set on this arc indicates that the subquery does not need information from the Select of node 15 to execute. Thus the arc connecting nodes 15 and 16 only needs represent the data-to-function flow of data; i.e., the set input to the Select operation.

7.6.2 Data Nodes with Multiple Children

In the basic tree representation only a function node can have multiple children. These children represent the parameters of the function represented by the node. In an extended representation, a data node can have more than one child. This representation allows for recording alternative methods for computing the same data. A similar approach is taken in other representations (e.g., [54], [121]), and is particularly useful when generating alternative execution plans. As a result, an optimizer has alternatives for representing object-creating operations — multiple occurrences of an object-creating operation in a query can be represented many times; can be represented as a single function node with multiple output occurrences; or can be represented as a single function with a single output object that is then shared by later operations. The multiple child representation is also useful in systems with run-time method binding since alternative bindings could be represented as different function nodes with, possibly, different annotations.

Determining values for annotations to a Data node with many children is an interesting problem, since different child functions could imply different annotations for the Data node. In general, if Data node annotations are derived from the characteristics of a single child Function node then all sibling function nodes must imply the same annotation value. For example, suppose an annotation is used to indicate data ordering. If a function produces a sorted result, the data node would be annotated to indicate sorting. Alternative functions producing this data node would also have to produce a sorted result.

To do cost-based optimization, an optimizer may need to consider that data could be produced by alternative functions having different costs. Cost of a data node could be, for example, the minimum of the costs of functions creating it.

All of these examples indicate that computation of annotations on data nodes is not independent of the representation choice. The semantics of an annotation at a data node could depend on whether or not the node can have multiple function children.

7.6.3 Graphical Representation

The query representation becomes a graph when a data or function node is allowed to have more than one parent. The reason for allowing multiple parents for a node is to manage the use of common data and expressions in a query. A data node with multiple parents indicates that the data is shared by different functions. For example, in the query of Figure 7.4 the People dataset is represented by two distinct nodes, 11 and 18, as input to two different Select operations. In a graph representation for the query, the People dataset could be represented by a single node connected

by an arc to the Select of node 10 and by another arc to the Select of node 17.[6] Such connections could be useful in determining an execution plan for a query. For example, an execution plan might scan the People dataset only once, processing each object in the set through both Select operations.

In this example the shared data node represents a dataset that is a constant to the query (i.e., the named dataset People). Although sharing of data nodes seems to offer advantages for query constants, it does not seem to provide any advantages for query variables. This is because variables are supplied by ancestors, and two data nodes representing the same variable must have a common ancestor. Since the source of the variable is already common, there seems little need to explicitly represent the sharing of that data variable through multiple parents of one node.

Interior data or function nodes can be shared, giving two possible ways to represent common subexpressions. Sharing of data nodes indicates that different functions have, as some argument, identically the same object. Of course, as noted in Section 4.4, determining whether two objects in a query are identical can be a difficult problem. In general, though, common subexpressions will be represented by sharing data nodes.

Two data nodes sharing the same function node indicates that a function executes once and produces two non-identical objects as its result (i.e., copies). For example, this is the case whenever a query is executed in a language like EQUAL that only supports objects. Each time a query expression is executed it produces a new object. Thus, although two expressions are identical as strings, and may even execute on the same database state, they always produce non-identical objects. Allowing function nodes to have multiple parents allows such a representation of object identity in the query representation.

The annotations on multiple parent arcs of data or functions nodes are related in that the Use sets on those arcs contain the same variables. The Avail sets on those arcs may be different, but in all cases Use must be a subset of the intersection of all the Avail sets.

In a graphical query representation there must be a single data node as the root of the graph. This node represents the result of the query. This node may have more than one child, indicating that the result can be computed in more than one way, and may even have more than one parent edge, indicating that the result of the query is used in more than one place in a program. A subquery is also a graph rooted at a single data node. If it has more than one parent edge, the subquery represents a common subexpression.

7.7 Summary

The internal query representation presented in this chapter generalizes operator trees by representing query operations, inputs, and predicates uniformly and by providing a means for annotating components with information about the query. Our representation supports the manipulation of query expressions nested in predicates of a query operation by representing those expressions in the same way as any other query expression. We also discussed Def-Use query annotations that support the processing of nested queries by storing information about variable usage. This type of information is used in the Join conversion region (CJ) in the example of the next chapter.

The representation and the annotations are both extensible and thus support the extensibility of the optimizer. The representation can be used to express any function, with any number and kind of input parameters, and thus adapts immediately to new algebraic operations in a model. The annotation tables for data, functions, and relationships (arcs) can be extended with new annotations to support the processing of new regions that may be added to an optimizer. Since

[6] Note that one reason for annotating the arcs (as opposed to nodes) with Avail and Use sets is to allow for such multiple connections.

types are represented as annotations, the representation also adapts immediately to the addition of new types to a model.

Annotations also provide a place for representing a query cost model. Information about data and operations needed by the cost model is stored as annotations to nodes, and methods of the model can use this information for computation. Cost methods can also update annotations representing query cost.

Chapter 8

An Example Epoq Optimizer

In this chapter we illustrate the capabilities of the Epoq architecture with a design for a simple Epoq optimizer. This design incorporates a number of strategies for rewriting object-oriented queries, one of which (CJ, see Section 8.2.1) is presented for the first time here.

The optimizer described here is an example rewrite system for EQUAL queries. We assume queries are represented as in Chapter 7 and we assume the existence of a simple cost model maintaining information about query cost as annotations to the representation. This optimizer is not comprehensive – it only incorporates strategies for rewriting queries. However, using the Epoq approach this optimizer could be used, for example, as a region for a higher-level optimizer. The higher-level optimizer could incorporate a control that uses other regions to do such tasks as translating a declarative query to the algebra and manipulating the result of algebraic rewrite to produce a query plan.

Although we present a region that represents a new strategy for processing object-oriented queries, the purpose of this example is not to propose new optimization strategies or to define new optimization heuristics. Indeed, designing good optimization strategies, and heuristics about interactions between different optimization strategies, is an open area of research. The Epoq architecture is a vehicle for defining such interactions and can be a testbed for experimentation with optimizer design heuristics.

In the following sections we specify an Epoq optimizer containing nine regions, two of which are implemented using the planning-based control of Chapter 6 and the rest of which are leaf regions. In particular, we specify the interface for leaf regions and the control for the interior regions. Before presenting the optimizer, we give a predicate language for specifying applicability and also define some region goals in Section 8.1. We present the overall optimizer design in Section 8.2, specifying the leaf regions in Section 8.2.1 and the root control in Section 8.2.2. We illustrate the extensibility of Epoq and the planning-based control by adding two new regions to the optimizer in Section 8.3. In Section 8.4 we review the main features of the architecture illustrated by this example.

8.1 Some Preliminaries

8.1.1 Applicability predicates

Applicability predicates are used in the left hand sides of planning rules to describe queries to which the rule may be applied. Applicability predicates are also provided at the interface by a subordinate region to describe the queries to which the subordinate may be applied. These predicates are called static applicability predicates because they can be evaluated in the presence of only the query, without any optimizer execution context. Recall that these predicates are not expected to exactly

characterize queries, but to give hints about queries the rule, or region, may be able to successfully process.

For this example, an applicability predicate P over a query is a first order calculus expression with a single query variable Q. In order to access the components of a query, we define the following set-forming functions. These functions take a query and return a set containing components of the query. In most cases, the query components returned are nodes of a query representation. These nodes are treated as objects. This allows distinctions between multiple occurrences of the same operator or data in a query.

- Ops(Q) – returns a set of the function nodes in query Q representing EQUAL query operations
- Root(Q) – returns the root node of query Q
- Data(Q) – returns a set of the leaf nodes representing database objects referred to in query Q
- Vars(Q) – returns a set of the names of the variables defined in the expression of query Q
- Paths(Q) – returns a set of the path expressions in query Q
- Preds(Q) – returns a set containing operations used in any predicates of queries in Q (e.g., Exists, Forall, $\wedge$, etc.)

Additional set-formers could be similarly defined as needed. Sets can be combined using any query operations, although union, difference and intersection will be most useful for our example; membership and empty predicates are also defined for sets. Equality for set operations is based on node identity (or value, where appropriate).

Applicability predicates are first order calculus expressions, with the qualification that all quantified variables are defined in terms of the form

$$[\exists/\forall] v \; v \in \text{Set}(Q)$$

where Set(Q) is an expression formed using the set-forming functions and set operations noted above. Sets can also be explicitly denoted using braces (e.g.{Ojoin} is a set containing the operator label "Ojoin").

For example, if Q is a query, some useful predicates are:

- $(\forall \; o \in \text{Ops}(Q))(o.\text{label} \in \{\text{Select, Project, Ojoin}\})$ — tests that query Q has only Select, Project and OJoin operations
- $(\exists \; x,y \in \text{Vars}(Q))(x \neq y)$ — tests that query Q has more than one variable defined
- Root(Q).label = Select — tests that Q is a Select query

These predicates and set-formers allow testing the existence of components of a query, but do not allow testing the structure of the query expression. For example, this applicability language cannot describe a test to find an Ojoin operation with a Select operation as its second input. This language seems suitable, though, for the applicabilities that need to be described in our example optimizer. Designing an applicability language that can query the tree structure of a query is an interesting research topic.

8.1.2 Goal specification

For the purposes of this example we assume that goals are instances of an object type called Goal. Individual instances of the type have unique names, and we use these names to access the goal.

Type Goal defines a Boolean function called *Success?*. For goal G, Success?(G) takes two query parameters: an (optional) initial query and a final query. The implementation of function Success?(G) for any G checks the state of the final query, or the relative states of the initial and final query, to determine whether the goal has been met.

The Success? function implements the same test for many goals: the goal is achieved if the (estimated) cost of the result query is lower than that of the initial query. In the example in this chapter, this is the success test for the following goals:

- Good_Rewrite
- Fast_Rewrite
- Join_Reorder
- Lower_Cost
- Predicate_Reorder
- Best_Est_Cost

The goals have different names because the semantics of the names indicate how a query is changed to effect the goal. For example, the Join_Reorder goal uses cost as a determination of success but implies by the name that the cost is reduced through finding an alternative, better join ordering. On the other hand, the goal Fast_Rewrite implies that heuristics will be used to lower the cost quickly. The success of the goal does not test the speed of the processing, however.

Other goals used in this example optimizer are:

- Convert_Join_Predicate: Success? = output query has a join operation that is not explicitly present in the input query
- SPJ_Form: Success? = output query is in a normalized SPJ form
- Simple_Transform: Success? = output query is not the same expression as the input query

Some of these success tests can be more easily implemented by not actually looking at the query. For example, the success of the Convert_Join_Predicate goal can be determined in the region working toward that goal by keeping track of whether appropriate transformations are made to the query. In the case of such knowledge, a region may choose not to use the formal Success? method to determine whether its goal is achieved.

8.1.3 Query Representation and Annotations

We assume queries are expressed in EQUAL and stored using the EAT representation of Chapter 7. The representation is annotated with information that will be used by different of the regions in the optimizer.

We also assume the existence of a simple cost model that can statically assess a query and provide an estimate of the processing cost for the query. This cost information is maintained as annotations on data nodes (i.e., cost is the estimated cost to produce this data) and function nodes

(i.e., cost is the estimated cost of executing the function). A cost method can, given a query, apply the cost model to the query to update the cost annotations.

Data nodes will be annotated with cost and type information. The type information is not directly used by any of the regions in the example optimizer, but could be used, for example, by a new region that does semantic query optimization [101]. Regions that do not use type information may add nodes that aren't type annotated. However, since EQUAL is strongly and statically typed, the correct type information can be generated as needed by an Annotate_Type method over the representation.

Function nodes are annotated with cost and def information. Cost is maintained, as noted above, by the cost method over the representation. Def information is stored when the query representation is initialized with a new query, and whenever operations are modified or added through transformations.

Function nodes also have an annotation called "kind" having type String. Kinds give more specific information about the query function they annotate. In particular, kind is used to annotate function nodes representing Ojoin operations to indicate whether they are the standard 2-way (commutative) joins as defined in EQUAL or whether they represent Left Outerjoin operations [122]. The Left Outerjoin operation

$$\text{LOJ(As,Bs,}\lambda a, b\ p(a,b))$$

(called hereafter just Outerjoin) is equivalent to the following EQUAL Project operation:

$$\text{Project(As,}\lambda a\ \ <\text{(A,a), (B, Select(Bs,}\lambda b\ \text{p(a,b)))}>)$$

Use of the Outerjoin operation provides a number of useful transformations for our example optimizer. Some nested Project operations can be transformed to Ojoin and sometimes Outerjoin operations, and there are a number of transformations for join and outerjoin operations (e.g. [38], [50]). Thus, rather than using the kind=outerjoin annotation on an Ojoin function node, we extend the algebra with an Outerjoin operation. The explicit operation, as opposed to the annotated Ojoin, will be easier to recognize and manipulate in transformations.

8.2 An Optimizer Design

The example optimizer consists of nine regions connected as depicted in Figure 8.1. Each box in the picture is a region and is labelled with a name for the region as well as the name of each region's goal(s). The root region, named OPT, uses the following regions to do its transformations:

- SX does simple query transformations(X) such as predicate simplification, view substitution, collapsing Project operations, etc.
- CJ converts nested predicates to Join operations, when possible.
- OJ reorders join operations. In particular, this region can handle OuterJoin and Join operations.
- SPJ tries to convert nested queries involving Select, Project and Join operations into a canonical form with all Joins followed by Selects followed by Projects. Such a form could be conducive to lower level query optimizations.
- DP reorders join operations using a dynamic programming algorithm and a simple cost model.

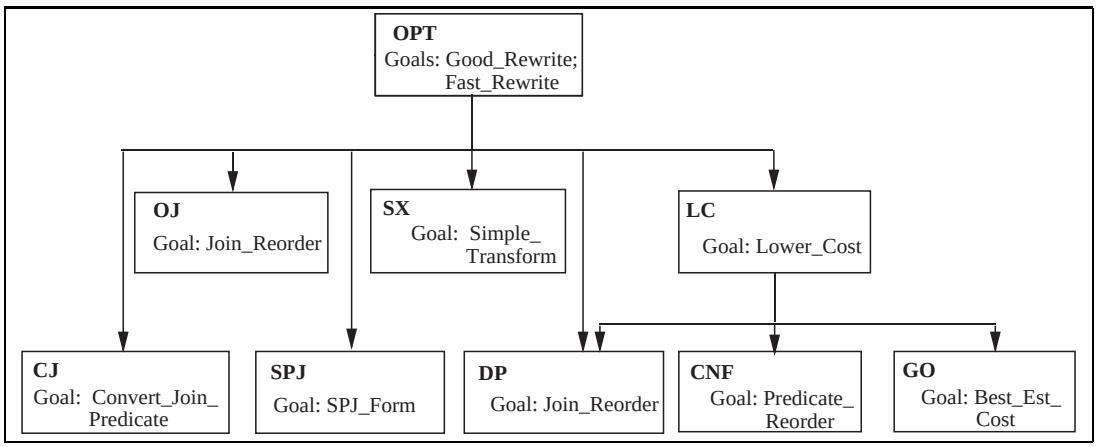

Figure 8.1: Example optimizer design

- LC tries to lower the expected cost of the query.

The OPT region takes a query in a high-level algebraic language and applies its subordinate regions to manipulate the query and produce an algebraic query with lower expected cost. OPT can use its subordinates in different ways depending on the nature of the query it is trying to optimize. For example, a simple query with no nested expressions could be simply processed by region LC to lower the expected cost. More complex queries, with nested expressions, can be processed by a region that works to unnest the expressions (e.g., SPJ or CJ) followed by a region that reorders the resulting Join operations (OJ or DP).

Region OPT can also choose to process queries using a "pilot pass" style algorithm [119] that first applies simple transformations to the query (region SX) and is satisfied with the result (and quits) if those transformations reduce the expected cost by some amount. If the preliminary pass is not satisfactory, more complete processing of the query can be undertaken by the lower cost region (LC), for example. Such an algorithm could speed processing of simple queries by the optimizer.

The lower_cost region (LC) takes advantage of the hierarchical structure of the Epoq approach. This region has three subordinate regions which it can use in trying to achieve its goal of lowering the expected cost of its input query expression. Region GO is a generated optimizer module [58] that uses transformation rules to do local rewrite of EQUAL queries. Region LC can choose to use this region to attain its own goal, or can choose to send the query through a sequence of modules (CNF and DP, here) each of which has its own strategy for applying its transformations to the query (similarly to [45] or [129]). LC could even try both strategies, choosing the best result, or could use the strategies in a pilot pass type of approach.

8.2.1 Region Descriptions

In this section we briefly describe all of the regions in the optimizer, except the root region. We give interface descriptions for these regions, and discuss the behavior of each region. We also describe the planning-based control implemented for the LC internal region. The root region has an interface to a higher-level query processing system, which we will not describe, and has a planning-based control that is described in Section 8.2.2.

Most of the leaf regions are patterned after rewrite techniques for relational, complex object, and object-oriented systems that appear in the literature. Region CJ, though, represents a new

Region	*Goal(s)*	*Ap(Q)*
CJ	Convert_Join_Predicate	$(\exists o_1, o_2 \in \text{Ops(Q)})(\exists p \in \text{Preds(Q)})$ $((o_1.\text{label} = \text{Image} \wedge o_2.\text{label} = \text{Select}) \vee$ $(o_1.\text{label} = \text{Project} \wedge o_2.\text{label} = \text{Select}) \vee$ $(o_1.\text{label} = \text{Select} \wedge \text{p.label} = \exists))$
OJ	Join_Reorder	$(\exists o \in \text{Ops(q)})(\text{o.label} = \text{LOJ})$
SPJ	SPJ_Form	$(\forall o \in \text{Ops(Q)})(\text{o.label} = \text{Select} \vee$ $\text{o.label} = \text{Project} \vee \text{o.label} = \text{Ojoin})$
SX	Simple_Transform	True
DP	Join_Reorder	$(\exists o_1, o_2 \in \text{Ops(Q)})$ $(o_1 \neq o_2 \wedge o_1.\text{label} = o_2.\text{label} = \text{Ojoin})$
CNF	Predicate_Reorder	Not Empty(Preds(Q))
GO	Best_Est_Cost	True
LC	Lower_Cost	True

Table 8.1: Region Interface Specifications

strategy for the manipulation of object-oriented queries. We will therefore discuss this region in more detail than the others.

The static interface specifications for each of the leaf and interior regions are detailed in Table 8.1. As noted in Chapter 6, the static interface consists of a statement of goals a region can try to attain, and an applicability predicate characterizing queries the region will process. In the table, this information is indexed by the name for the region.

Region SX – Simplifying Transformations

This region takes as its input any query (indicated by Ap(Q) = True in Table 8.1) and tries a number of transformations to reorder, and possibly simplify, the query for future processing. The region does not consider the estimated cost of the query it is transforming, and is successful when it is able to perform some transformation on a query. It chooses to return the transformed query for which the size of Ops(Q) plus the size of Preds(Q) is minimum.

Transformations performed by this region include view substitution, predicate simplification (e.g., removing clauses that are always true; applying transitivity), collapsing Project operations, collapsing Select operations over the same set, changing predicates with OR operators to Unions (when appropriate), etc. These transformations may provide all the processing that is possible for very simple queries (such as single, unnested Select operations). On the other hand, for more complex queries the application of such transformations may find a query that is conducive to further processing in the optimizer. These heuristics are used in rules of region OPT when choosing to use SX (see Section 8.2.2).

Region CJ – Convert to Join

Region CJ is designed explicitly to work with nested EQUAL queries. In particular, it can recognize special predicates in Select operations that are nested inside Image or Project operations. Such predicates can be translated to give flat queries using Outerjoin (and occasionally Join) operations. If translated to SQL-like queries, the nested queries would appear in the Select clause of the query. Thus, this region complements recently proposed techniques for dealing with SQL-like queries with nested queries in From and Where clauses [31, 38, 109, 115].

The applicability predicate for this region states that it looks for queries with Image and Select operations, Project and Select operations, or Select with Exist operations. In particular, the region can transform Selects nested within Images or Projects, and Selects with an existence predicate referring to a different set than the one over which Select ranges.

The region executes by making two passes over a query represented as in Chapter 7. The first pass searches for patterns indicating that the query can be converted into join operations, and manipulates Select operations to move them into the correct locations for further transformation. This pass uses Def, Avail and Use annotations on the query tree in its search for join predicates and "movable" Select arguments. If the pass is successful, a second pass actually transforms the query [148].

The region stops after it makes a single conversion (or if it cannot convert), thus should be executed repeatedly by a parent region until there are no more conversions to be done. Also, this region can sometimes find more conversions to do if it is alternated with a general query transformer such as region SX or GO. Region CJ performs some directed transformations that move Select operations into particular conformations. Sometimes these transformations will not apply because other operations obscure the required patterns. Thus a region that does transformations such as moving Selects to leaves, and moving Projects toward the root can often expose patterns that can be used by region CJ. The control of OPT will take advantage of this heuristic.

Region CJ provides a dynamic applicability predicate at its parent interface that can determine exactly whether the region can transform the query successfully. This predicate is executed in region CJ, on request from the parent region (OPT), and uses information from the first pass search to make its determination. The dynamic applicability function is not used in this example because no other region duplicates the goal of CJ.

The goal of this region is Convert_Join_Predicate. The region knows it is successful at achieving the goal when it applies the converting transformations to the input query (i.e., when it completes the second pass).

Region SPJ – Normalize Select-Project-Join Queries

The goal of this region is related to the goal of region CJ – nested queries are transformed, as much as possible, into a normalized form where all Join operations will be executed followed by Selects, then Projects. This is similar to the goals of the Starburst rewrite system [115] and this region can be built based on their ideas.

This region unnests a query as much as it can, converting a query to an S-P-J normal form. In other words, subsequent executions will not further process a query. Also, the region is complemented by regions that do join reordering. This information is used when building planning rules in the parent region.

Region OJ – Outerjoin transformations

The goal of this region is Join_Reorder, but it specifically reorders queries involving joins and outerjoins. This is indicated by the applicability predicate that states the region is interested in queries with outerjoin operators.

This region can use the transformations of Galindo-Legaria and Rosenthal [50] to simplify and reorder queries. Their algorithms require queries be represented as operator trees; the internal representation of our example optimizer meets this requirement. The region can detect success when it successfully applies a reordering transformation to its input query.

Outerjoin transformations are useful after the application of region CJ, since it converts nested predicates to join and outerjoin operations. This relationship between the regions is defined by rules in the Good_Rewrite goal package of the root region (see Section 8.2.2).

Region DP – Reorder Joins

Region DP also reorders join operations, but cannot deal with outerjoins. Both this region and region OJ have the same goal, so the applicability predicates of the regions make the distinction between OJ, which can process outerjoins, and DP, which cannot. This distinction will be detected when the applicability predicates are tested in the primitive goal execution of region OPT, and the appropriate region will be chosen to execute depending on the joins present in the query. The applicability predicate of region DP states that there must be at least two join operations for the region to process a query.

Region DP uses a dynamic programming approach to reorder join operations. In order to apply this approach, the region uses the cost annotations on the query and cost model operations to compute costs of reordered operation sequences. The region considers itself successful when the expected cost of its result query is lower than the expected cost of the input query.

In order to process successfully, the join operations must be clustered in the query. Thus, this region should be preceded by a region such as SPJ that produces a query with clustered joins.

Region CNF – Reorder Predicates

The goal of the CNF region is to simplify the predicates of a query. This is similar to the predicate simplification done by region SX. The difference here is that region CNF first manipulates a query into an SPJ normal form (if possible), converts the Select predicate to conjunctive normal form, then simplifies the CNF predicate. This region is patterned after the pre-processor module of Sieg [137] in conjunction with his boolean expert module. We implement these two modules as a single region for simplicity of explanation. They could also be naturally implemented (as Sieg does) as a parent (pre-processor) and child (boolean expert) region.

This region should be followed by a region that finds good join orderings. This heuristic appears in the control rules of its parent region (LC).

Region GO - Generated Optimizer

This region was generated using the EXODUS optimizer generator [54]. It applies EQUAL rewrite rules to general queries (including those given on page 41, for example) using a simple, internal cost model to estimate query cost and direct the transformation process. The interface required by OPT is wrapped around the resulting optimizer to make it a region. This includes a goal statement indicating a goal of Best_Est_Cost, and an applicability predicate accepting all queries (Ap(Q) = True).

This approach can be used to build other regions that are rewrite systems based on rules. The disadvantage to this approach is that the generated optimizer has its own internal query representation and cost model. Although the representation is still an operator tree, the conversion to and from the EXODUS representation is time-consuming and loses annotation information. The advantage of this approach, of course, is the ease of building a rule search system. We do not use EXODUS to generate any of the other regions in this example.

GOAL PACKAGE Lower_Cost QUERY Q
SEARCH give priority to most matching clauses ∧ one success per rule TERMINATION no rule applies
LOCAL STORE OBJ Q_0, Q_1: Query ← Q RESULT Q ← choose lower est cost (Q_0, Q_1)
METHODS NOT EMPTY (Preds(Q)) −→ ACHIEVE Predicate_Reorder ON Q GIVING Q_0; ACHIEVE Join_Reorder ON Q_0. −→ ACHIEVE Best_Est_Cost ON Q GIVING Q_1.

Figure 8.2: Lower_Cost goal package.

Region LC – Interior Region for Lowering Cost

Region LC is built as an interior region controlling the execution of subordinate regions to try to lower the cost of its input query. The applicability predicate for LC states that it can accept any query, and it will be successful in its transformation if it produces a result query with lower estimated cost than the input query.

As an interior region, LC is built with the planning-based control of Section 6.4. It controls three subordinate regions. Regions CNF and DP are designed to work in sequence to put a query in a normal form, manipulate the predicate and then find a good ordering for join operations. This will work well for relational style queries for which there are complicated predicates, so can be executed if Preds(Q) is not empty (indicating there is at least one predicate operator). Region GO is a generated query rewrite system. It works on any query to find a query of lower estimated cost by applying transformations to the EQUAL expressions. For the purposes of this example, LC tries these two tactics, and chooses the best of the two results. This strategy is specified in the rules of the goal package of Figure 8.2.

The condition on the first rule (NOT EMPTY (Preds(Q)) indicates that the rule is only useful if the query has any predicate operators. If there are predicates this rule will always be executed first, since one search criterion is that the rule with most matching clauses has priority. If the first rule has been executed, the second rule will then be executed by default, since it applies to any query and the search strategy will not apply the same rule to the same query twice. Since the package, by definition, always works with query Q, and none of the rules modify Q (i.e., all transformations and results are stored in new query variables), the region will execute each rule at most once.

If a rule execution successfully completes, the result will be stored in the local memory. The first rule stores its result as Q_0; the second rule stores its result as Q_1. Both of these variables are initialized with the input query (in the OBJ declaration). The final result is chosen using the estimated costs of the Q_0 and Q_1 results.

This is a very simple region control that illustrates the use of rules and search strategy to model choice of alternative executions. This region also illustrates the use of the local store to maintain temporary processing results. This is similar to, but much simpler than, the management of query choices in an optimizer's global store.

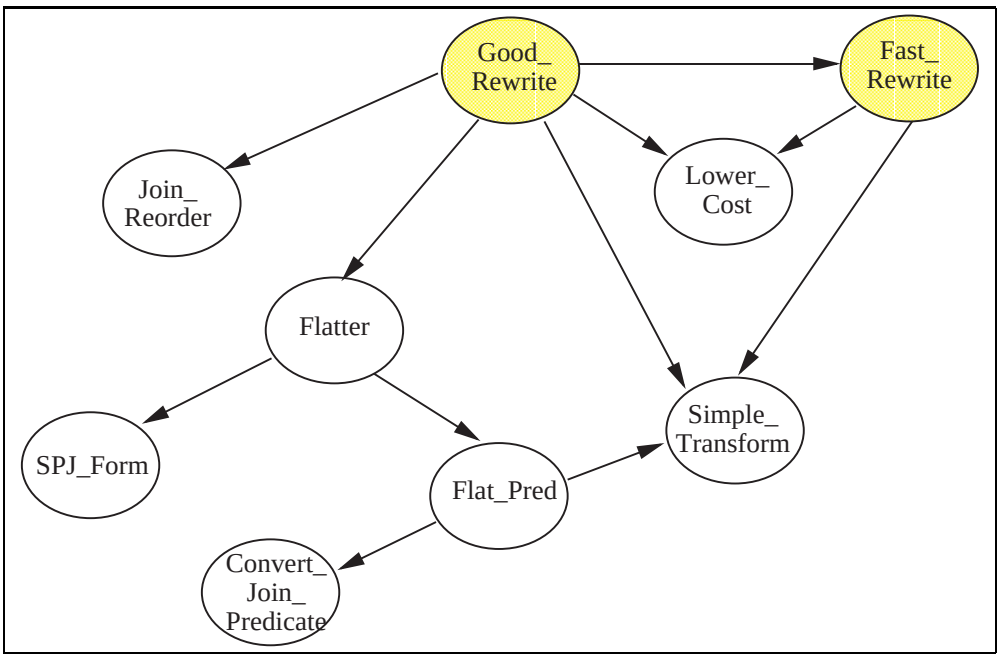

Figure 8.3: Relationships Between Goals in OPT

8.2.2 Root Region Control

The root region of the optimizer controls the transformation of a query supplied by an external processing system. The external system supplies a query, and chooses one of the goals that can be attained by the root region. As can be seen in Figure 8.1, the root region can attain either of two goals: Good_Rewrite and Fast_Rewrite. We assume that an external system calls the optimizer root module and passes it a query and one of those goals.

We assume for the purposes of this example that the optimizer starts with a single query and goal, and transforms that query successively until termination. In other words, we will not consider alternatives such as processing goals on subqueries. As we noted in the explanations of the subordinate regions, some of these regions may process a single query in more than one way and make a choice of result. At the root level, however, we assume a single path of transformations from input query to result. This simplifies the presentation of the example, but does not detract from the usefulness and extensibility of the control. Matching subqueries and goals, for example, could be accomplished by a separate module that searches through the queries in global memory looking for good possibilities for processing subqueries. Also, rule statements could be augmented to make suggestions for alternative processing by inserting (Goal,Query) pairs into the global memory. We leave the exploration of such strategies to further research.

The goals used in processing in region OPT are:

- two region goals: Good_Rewrite and Fast_Rewrite.
- five primitive goals: Simple_Transform, Lower_Cost, Join_Reorder, SPJ_Form, and Convert_Join_Predicate.
- two additional planning goals: Flatter and Flat_Pred.

The goals are related to each other as indicated in Figure 8.3. The nodes in the graph represent goals, and arcs are directed from a goal on the left hand side of a rule (i.e., a goal described by a

GOAL PACKAGE Good_Rewrite QUERY Q SUCCESS Success?(Good_Rewrite, QUERY, RESULT)
SEARCH priority by rule number $\wedge$ one success per rule TERMINATION no rule applies
RESULT Q
METHODS 1. $((\forall x, y \in \text{Vars(Q)})(\text{x} = \text{y})) \wedge ((\exists o \in \text{Ops(Q)})(\text{o.label} = \text{Select}))$ $\longrightarrow$ ACHIEVE Simple_Transform ON Q. 2. $(\forall x, y \in \text{Vars(Q)})(\text{x} = \text{y})$ $\longrightarrow$ ACHIEVE Fast_Rewrite ON Q. 3. $\exists x \in (\text{Ops(Q)} - \{\text{Root(Q)}\})$ $\longrightarrow$ ACHIEVE Flatter ON Q. 4. $(\exists o \in \text{Ops(Q)})(\text{o.label} = \text{Ojoin} \vee \text{o.label} = \text{LOJ})$ $\longrightarrow$ ACHIEVE Join_Reorder ON Q. 5. $\longrightarrow$ ACHIEVE Lower_Cost ON Q.

Figure 8.4: Good_Rewrite goal package.

package) to goals on the right hand sides of rules (i.e., goals used in achieving the package goal). In other words, the arcs point from tasks to subtasks. The shaded nodes represent attainable goals for the OPT region. Note that this graph is acyclic. This represents the fact that the rules are not recursive.

The execution of the optimizer uses the attainable and planning goals to plan a sequence of executions of primitive goals. The planning process is described by the goal packages of Figures 8.4, 8.5, 8.6, and 8.7. Primitive goals direct the execution of subordinate regions. Most of the primitive goals are associated with a single region, so there are no decisions to be made about the execution. However, the primitive goal Join_Reorder is declared by two leaf regions, so achieving that goal involves decisions between the regions by the primitive execution module. This is discussed further in Section 8.2.3.

Package Good_Rewrite

The Good_Rewrite package accepts a query Q and manipulates the query to try to produce a lower-cost result. Success for the package is defined as the Success? function for the Good_Rewrite goal (see Section 8.1.2) applied to the input and output query. The package doesn't use success for termination, but applies as many rules as it can before terminating, and checking success.

The Good_Rewrite package selects a transformation strategy depending on characteristics of the current query. The rules in this package are defined in a priority order – the first rule that matches the input query will be the first rule tried. The package will set up a priority queue of rules that match the current query, apply rules in order until one is successful, then set up a new priority queue of rules that match the result. This process will continue for no more than five successful rule applications, since the search strategy will only allow one successful application per rule. If at any point in the transformation process no rules apply to the current query then, of course, the package terminates.

The rules basically divide the queries into simple one variable queries, and more complicated nested queries. Simple queries involving Select are first processed by a region that does Simple_Transform; other simple queries are processed with Fast_Rewrite. The general heuristic for

GOAL PACKAGE Fast_Rewrite
QUERY Q
SUCCESS one rule successful

SEARCH one success per rule
TERMINATION no rule applies

LOCAL STORE
TYPE Control_Flag: Record of (id: string, val: Bool, query: Query)
OBJ Q_1: Query
RESULT Q

METHODS

$\longrightarrow$ ACHIEVE Simple_Transform ON Q GIVING Q_1;
Lwm_Add(Done, (Control_Flag
new(id = rule1, val = True, query = Q_1))).

Lwm_Match(Control_Flag (id = rule1 $\wedge$ val = True $\wedge$ query = ?Q))
$\wedge$ (est_cost(?Q) $\leq$ .5 * est_cost(Q)) $\longrightarrow$ Lwm_Apply(Assign,Q,(?Q)))

Lwm_Match(Control_Flag (id = rule1 $\wedge$ val = True $\wedge$ query = ?Q))
$\wedge$ (est_cost(?Q) $>$ .5 * est_cost(Q)) $\longrightarrow$ ACHIEVE Lower_Cost ON Q.

Figure 8.5: Fast_Rewrite goal package.

nested queries is to flatten the query, then reorder the resulting join operations. This heuristic is captured in rules 3 and 4. Rule 3 flattens a query, if possible, and rule 4 takes results of rule 3 (or any query with join operations) and reorders the joins. Two rules are used, instead of one, because it is possible that a query can't be flattened but would still be conducive to join reordering. It is also possible that a flattened query may not be re-orderable.

As rules are applied, the characteristics of the query may change. Thus the package iterates until each rule has had an opportunity to match the query characteristics. For example, a query that satisfies rule 1 may be processed by Simple_Transform to produce a nested query (e.g., if views are substituted). This query could then be processed by one of the rules for nested queries. Conversely, a query processed by rule 5 to lower cost may possibly be reduced to a single variable query (e.g., simplifying unions and selects). That result can then be processed by rule 1 or 2 (or both).

Package Fast_Rewrite

The Fast_Rewrite goal package executes the following "pilot pass" style algorithm [119]:

1. Achieve Simple_Transform on Q Giving Q_1.
2. If Q has been improved by 50%, quit;
3. Otherwise Achieve Lower_Cost on Q.

The idea here is that the preliminary Simple_Transform application may quickly improve the query to a point which eliminates the need to perform more expensive processing of the query. If Simple_Transform can't improve the query enough, then Lower_Cost is applied to find the lowest possible cost for the query. The heuristic tries to speed up the processing of the optimizer itself – avoiding using the (assumed) slower Lower_Cost procedure when possible.

The package uses a flag in the local store to control the conditional application of the rules. Initially, there is no flag in the store so only the first rule, which applies the Simple_Transform goal, will match the query. The Simple_Transform application stores its result in a temporary variable, so the initial query is retained for possible processing by Lower_Cost.

GOAL PACKAGE Flatter QUERY Q SUCCESS one rule successful	
SEARCH give priority to most matching clauses TERMINATION one rule successful $\vee$ no rule applies	
RESULT Q	
METHODS	
	$\longrightarrow$ ACHIEVE SPJ_Form ON Q.
$(\exists o_1, o_2 \in \text{Ops(Q)})(\exists p \in \text{Preds(Q)})$ $((o_1.\text{label} = \text{Image} \wedge o_2.\text{label} = \text{Select})$ $\vee\ (o_1.\text{label} = \text{Project} \wedge o_2.\text{label} = \text{Select})$ $\vee\ (o_1.\text{label} = \text{Select} \wedge \text{p.label} = \exists))$	$\longrightarrow$ ACHIEVE Flat_Pred ON Q.

Figure 8.6: Flatter goal package.

The first rule sets a flag to indicate that the rule has completed. This signals the second and third rules. These rules are partitioned by whether or not the query has been improved by the required amount. If the query has been sufficiently improved, the temporary result is moved to Q where it is recognized as the package result. If the query has not been improved the third rule executes and modifies the input query. If the rule is successful, its output is the package result. If it is not successful, the package will return the original query Q.

Package Flatter

The desired processing for flattening nested queries is either 1) a single execution of region SPJ to produce a Select-Project-Join normal form or 2) a repetitive execution of the CJ region to search for and manipulate nested join predicates. Goal package Flatter is used to choose between the two alternatives. Each alternative is represented as a single rule in Flatter, and the iteration required in the second alternative is achieved by encapsulating it in another goal (Flat_Pred).

The choice between alternatives is made according to the characteristics of the input query. If the query meets the requirements for region CJ, then the Flat_Pred alternative is chosen. Otherwise, the SPJ_Form alternative is chosen. Of course, standard package execution will ensure that if one alternative fails the other will be tried.

Success for this package can be tested as successful application of either of the rules. This success check actually just relays the success indication of the primitive regions (CJ and SPJ) where success can be more easily and accurately determined.

Package Flat_Pred

The CJ region is the only leaf region in this example optimizer that does not process its input query to saturation. Other leaf regions process a query as completely as they can before returning control to a parent. The CJ region, however, processes one predicate-to-join transformation then returns. As a result, the parent control needs to repeatedly call the region until it does no more transformations. This repetition is handled by the Flat_Pred package.

GOAL PACKAGE Flat_Pred
QUERY Q
SUCCESS one rule successful

SEARCH give priority to most matching clauses
TERMINATION no more rules

LOCAL STORE
TYPE Control_Flag: Record of (id: string, val: Bool)
RESULT Q

METHODS
$(\exists o_1, o_2 \in \text{Ops(Q)})(\exists p \in \text{Preds(Q)})$
$((o_1.\text{label} = \text{Image} \wedge o_2.\text{label} = \text{Select})$
$\vee\ (o_1.\text{label} = \text{Project} \wedge o_2.\text{label} = \text{Select})$
$\vee\ (o_1.\text{label} = \text{Select} \wedge \text{p.label} = \exists))$ $\longrightarrow$ ACHIEVE Convert_Join_Pred ON Q.

$(\ (\exists o_1, o_2 \in \text{Ops(Q)})(\exists p \in \text{Preds(Q)})$
$\ \ ((o_1.\text{label} = \text{Image} \wedge o_2.\text{label} = \text{Select})$
$\ \ \vee\ (o_1.\text{label} = \text{Project} \wedge o_2.\text{label} = \text{Select})$
$\ \ \vee\ (o_1.\text{label} = \text{Select} \wedge \text{p.label} = \exists)))$
$\wedge$ (Lwm_Match(Control_Flag (name = ?N $\wedge$ id = flag1 $\wedge$ val = True)))
$\longrightarrow$ ACHIEVE Convert_Join_Pred ON Q;
Lwm_Del(Control_Flag ?N).

NOT Lwm_Match(Control_Flag (id = flag1 $\wedge$ val = True))
$\longrightarrow$ ACHIEVE Simple_Transform ON Q;
Lwm_Add(Done, Control_Flag
new(id = flag1, val = True)).

Figure 8.7: Flat_Pred goal package.

The rules of the Flat_Pred package describe the following algorithm:

1. Apply CJ region to saturation.
2. Apply SX region. If not successful, quit.
3. Repeat steps 1 and 2, in order, until first application of CJ in step 1 fails (or step 2 fails).

The CJ region returns success every time it is able to find a predicate and convert it to a join (or outerjoin) operation. The region is applied repeatedly until it is no longer successful. At that point, however, its failure may be due to the fact that there are intervening query operations obscuring transformations CJ can perform. Thus region SX is tried with the hope that it will perform transformations that expose new possibilities for CJ. If SX can perform transformations, then CJ is again applied repeatedly until no transformations are found. This process is repeated until SX cannot find a transformation to perform, or CJ cannot find a join predicate in its first attempt after an SX execution.

The rules implement this strategy using memory flags and the search criteria that the rule with the most matching clauses will have highest priority. As a result, the initial query will match the first and third rules, in that order, so the first rule will execute. The iteration of the goal package will continue executing the first rule until the query characteristics no longer match the rule condition or until the rule fails. At that point the third rule will be tried.

If there are no simple transforms to perform, processing is complete — no more rules apply and the package quits. If there are simple transforms to perform, the third rule will succeed and

set a memory flag. This flag is used to generate one execution of the second rule. This execution generates a single execution of region CJ. If the execution is not successful there is no longer a possibility of finding transformations — no more rules apply so the package quits. If the single execution of CJ is successful, the memory flag is removed and the iterative process repeats starting with the first rule.

8.2.3 Optimizer Execution

A query entering this example optimizer gets matched with the requested goal and stored as a (goal, query) pair in the global control store. For the purposes of this example, we assume no other module can enter (goal, query) pairs (e.g., there is no module searching for subqueries) so the high-level region execution module must only pass the query to the appropriate goal package. We also assume that no alternative queries are stored in global memory. In other words, the Add_Choice method is null. These choices simplify the presentation of the example.

A query with a requested goal of Good_Rewrite will be processed according to its characteristics. This processing was discussed in Section 8.2.2. In this region a query is matched to a rule according to applicability predicates in the rule. After a query is successfully transformed, it may match and be processed by different rules.

Achieve actions with non-primitive goals induce a forward chain through goals to find a primitive goals. The paths of Figure 8.3 indicate the possible goal chains. Depending on the rules in a package, a query can follow one or more paths from any goal. For example, a query in the Fast_Rewrite goal package will follow the path to Simple_Transform and may follow the path to Lower_Cost. A query in the Good_Rewrite package will always, eventually, follow the path to Lower_Cost.

Many of the rules use primitive goals, and thus immediately generate (through the primitive execution module) execution of a subordinate region. Since most regions have unique goals, the primitive execution module needs to only decide whether the region is applicable, and determine whether the region application was successful.

Primitive execution for the Join_Reorder goal has to make a choice, though, since there are two subordinate regions (OJ and DP) that can achieve the goal. That choice is made using the applicability predicates provided by the regions. If a query contains Outerjoin operations, it will match to region OJ. If it contains Join operations it will match to DP. However, if the query contains both kinds of operations, it will match the static applicability predicates of both regions. In order to deal with this situation, regions OJ and DP provide dynamic applicability functions. These particular applicability functions take a query and goal as input[1] and provide an integer from 0 to 10 as output. A 0 indicates that the region has discovered it cannot process the query; a 1 indicates minimal processing may be done and a 10 indicates the region may do a lot of processing.

The primitive execution module can request that each region execute the dynamic applicability function, and will provide the query and goal for execution. Each region will apply its function and return its response. The function for OJ, for example, scans the query looking for sequences of mixed Joins and Outerjoins. It returns the sum of the lengths of all sequences found (up to 10). The function for DP, on the other hand, scans looking for Join-only sequences and returns the sum of the lengths of the sequences found. For each of these regions, the assumption is that the significant processing is more likely to be done on longer, or many, sequences of operations. The region responses are used to decide which of the two regions will be executed.

[1] The goal is redundant, since each region only has a single goal. However, dynamic applicability functions require a goal parameter to handle the case when a region can work toward more than one goal (as can OPT for example).

8.3 Extending the Optimizer

The optimizer in this example ignores many of the problems that are encountered with object-oriented queries. For example, this optimizer design doesn't include any processing for path expressions (as in [91], for example) or any type specific optimization (e.g., [101]). Such optimizations could be included as leaf regions, and the control of OPT could use these strategies in conjunction with the other strategies when appropriate. For example, path expression processing might be combined with join/outerjoin reordering. The Epoq architecture allows such additions to the repertoire of strategies and the control provides for extensions that can incorporate these additions.

New regions can be added to an optimizer by choosing a parent (or parents) for the region, providing the appropriate interface for the region, and, if necessary, adding control rules to the parent region. In this section we add two new regions to the example optimizer. One region, which we call region J4 because it reorders sequences of four or fewer joins, illustrates the interface compatibility requirements. The other region, called PE because it processes path expressions, requires modifications to the parent control rules. We add both of these regions as leaves of region OPT.

8.3.1 Duplicating Goals

Region J4 offers alternative processing for join reordering. This region has a specialized control that can quickly manipulate short join sequences to find optimal orderings. The region goal is Join_Reorder and its static applicability function is

$$\text{Ap(Q)} = (\exists o \in (\text{Ops(Q)})(o = \text{Ojoin})$$

In other words, Ops(Q) contains Ojoin operations.

Since J4 duplicates a goal that is already in the optimizer, the existing rules do not need to be modified. The only modification to the optimizer is to update the Join_Reorder primitive action module to recognize the new region. Since the module actually uses a table containing information about the regions, only the table needs updating. When the module checks static applicability predicates, queries that satisfy the applicability predicate of J4 will be directed to that region.

The Join_Reorder primitive module also requires a dynamic applicability function. Recall that this function accepts a query and goal, and returns an integer from 0 to 10 evaluating the region's applicability to the query. The dynamic applicability function for region J4 scans the query looking for join sequences. It returns an evaluation of 10 if it finds sequences of length less than or equal to 4, and an evaluation of 0 if it finds longer sequences. These evaluations indicate that the strength of region J4 is in reordering sequences of four or fewer joins.

8.3.2 New Goals

Adding strategies with new goals to an optimizer requires that the optimizer control be extended to use the strategies. Suppose we want to add a new region that can handle path expressions. Such a region could be based on one of the strategies of [91], for example. The interface for the region declares a goal of Fast_Paths, and an applicability predicate of[2]

$$\text{Ap(Q)} = \text{NOT EMPTY (Paths(Q))}$$

[2]Note that we conveniently defined Paths(Q) as one of our set-formers in anticipation of adding this region. An ideal applicability language would be general enough to query over all situations that can occur with queries, and would be able to be easily extended to handle extensions to the query algebra.

The difficult part of adding a region with a new goal is determining how to incorporate it into the existing control of the optimizer. In this example, we need to determine where the Fast_Path region fits in relation to other regions in the optimizer, and how the control of a parent region can use the capabilities of Fast_Paths.

Suppose we decide that a good heuristic for managing simple queries with path expressions is to perform Simple_Transforms on the query, then improve the paths. This heuristic could be implemented by adding the following rule as rule 2a in the Good_Rewrite package:

NOT EMPTY(Paths(Q)) $\longrightarrow$ ACHIEVE Fast_Path ON Q.

This rule will follow the rules that perform simple transforms on single variable queries, since the search strategy puts rules in the priority queue by their order in the package. In addition, if a nested query has paths, the paths will be processed first, since the rule precedes the rules for nested queries.

An alternative (or additional) use for the Fast_Paths task is as part of the Lower_Cost region. If a query contains path expressions, it may be useful to process those expressions before the general rewrite of the query done by the Best_Est_Cost goal. It may also be useful to process paths after reordering the predicates and before reordering joins.

A problem with describing rules in Lower_Cost to handle these situations is that we do not necessarily want the failure of the Fast_Paths goal to cancel further processing. Whether or not the expression has path expressions, we still want to consider only the two alternative results generated by the current two rules.

A simple way to handle this problem is to define an additional subgoal, called Preprocess. The package for this goal contains a single rule:

NOT EMPTY (Paths(Q)) $\longrightarrow$ ACHIEVE Fast_Paths ON Q.

which is executed once, if applicable. The package always returns Success = True, and will return either the result of achieving Fast_Paths or, if Fast_Paths doesn't apply or fails, the original query.

Now, the rules in the Lower_Cost goal package (Figure 8.2) can be replaced with the following two rules:

NOT EMPTY (Preds(Q)) $\longrightarrow$ ACHIEVE Predicate_Reorder ON Q GIVING Q_0;
ACHIEVE Preprocess ON Q_0;
ACHIEVE Join_Reorder ON Q_0.

$\longrightarrow$ ACHIEVE Preprocess ON Q GIVING Q_1;
ACHIEVE Best_Est_Cost ON Q_1.

Execution of the Lower_Cost package proceeds as described in Section 8.2.1. The Preprocess steps have no effect on the package execution, other than the call to the Preprocess goal package. Preprocess will not affect the success or failure of either rule, since it will improve paths if possible and pass either the improved result or an unchanged query along to the next step of each rule.

8.4 Summary

The optimizer design in this chapter is intended to illustrate the capabilities of the Epoq approach. We chose as regions some common optimization strategies and showed how they could be simply characterized using the Epoq interface. We used some simple heuristics describing desired interactions between regions to illustrate the planning-based control.

The planning-based control can express sequential, iterative, and conditional processing of rules and, therefore, of region applications. The language expresses sequence through the right-hand sides of rules, or through the use of memory flags (e.g., as in the Fast_Rewrite goal package). Iteration can occur in the high-level control[3] and in package control. The high-level control iterates through (goal,query) pairs, and the package control iterates through rules. Conditionals are implemented using the query and memory conditions on the left hand sides of rules.

This example illustrates the ability to integrate a variety of optimizer processing strategies using the Epoq approach. The leaf regions in our example all implemented different strategies for performing their task. These strategies were integrated by the planning-based control of the Epoq control regions.

The common interface also allowed us to integrate modules from different sources. For example, the GO region is a rule-based rewrite system generated by the EXODUS optimizer generator. This region has its own internal cost model and query representation. Region CJ, on the other hand, was designed expressly for Epoq and takes advantage of the annotated query representation.

Finally, we illustrated the extensibility of an Epoq optimizer by adding new regions to the example. One region duplicated an existing goal in the optimizer, so required minimal integration. The other represented a new goal for the optimizer so required modification to the parent region control. The extensibility of the planning-based approach to control limited that modification to changes to planning rules.

[3]This capability was not illustrated in this example though.

Chapter 9

Summary and Future Directions

In this thesis we described an approach to query processing that supports the extensibility inherent in object-oriented databases. We summarize the main points of our approach in this chapter. We conclude with a discussion of some of our ideas for future research that expands on the work presented here.

9.1 Summary

We presented our work in the context of a general framework that describes query processing as the integration of a data model, a query language, transformation rules for the language, an internal query representation, a cost model, and a process for query optimization. We presented designs for components of the framework, and described the requirements for the different components to support each other and interact to process a query.

Our approach focussed on extensibility as the major challenge in supporting query processing in object-oriented database systems. We used the ENCORE model as the basis model for this work. Extensibility in the model is based on support for abstract data types. This extensibility places a requirement on the other components of the query processing system to support the additional types and operations that can be defined in the model, as well as their associated implementations.

The EQUAL algebra supports extensibility by accessing objects exclusively through their interface. Thus, extensions to the collection of abstract data types and associated operations, and modifications to data type implementations, are transparent to the algebra. In addition, the algebra is defined as methods over Set types in the model. Thus, the algebra itself could be extended by defining new methods for type Set.

We presented a new approach to providing extensibility in query optimization. The Epoq approach provides for extending the collection of processing strategies used to optimize a query. A processing strategy is encapsulated in a region. The Epoq architecture describes a common interface for a region, and in doing so characterizes processing strategies. The common interface abstracts the input/output characteristics of the strategy from its implementation, and thus allows for extending the optimizer with arbitrary new strategies that can manifest the appropriate I/O characteristics.

The interface includes an internal representation for queries that is also extensible. We defined extensible annotated trees (EATs) as a representation for queries. These trees uniformly represent query operators, query predicates and user-defined methods, and naturally integrate changes in the user-defined collection of abstract data types. We also allow extensions to the annotations on components of the representation to provide for new information about query components that

may be required, or generated, during query optimization.

We described a planning-based control that integrates the processing of different regions to process a single query. Whereas optimizers normally plan the execution of a query, our control additionally plans the optimization process for a query. The control determines what processing to do depending on the characteristics of the query and the processing that has already been performed on the query. The control is a rule-based planning system, where rules describe heuristics about the goals to be achieved to optimize a query. These goals are used to plan the actions performed by the optimizer. This control is also extensible and thus supports the addition of new regions to an optimizer through extensions expressed in the control rule language.

Our support for extensibility in query processing leads naturally to support for the integration of independent system components. A cost model can be integrated into the system through extensions to the annotations on the query representation. New processing strategies can be integrated into an optimizer as new regions, or even as new control strategies defined using our planning-based control.

We illustrated the architecture with an example optimizer. We defined an example applicability language over query expressions and used this language to describe the input characteristics of the example regions in the optimizer. The example incorporated, as leaf regions, a variety of processing strategies proposed in the literature. We defined goals and rules for a planning-based control that integrates these strategies. We illustrated the extensibility of the Epoq approach by adding two new regions to the example optimizer.

9.2 Contributions of This Thesis

We described components of a framework for query processing that were designed with the requirements of the object-oriented database in mind. The development of these components included the following major contributions to understanding and solving the problem of processing queries in an object-oriented database.

- the EQUAL query algebra (Chapter 3).
  - an illustration that relational style query processing can be performed in object-oriented databases.
  - operations to create objects with unique identities and to manipulate the logical structures implied by object identities.
  - transformation rules for the algebraic operations.
  - a theory of query equivalence in the presence of object-creating operations.
- an internal query representation (Chapter 7).
  - an extensible representation for queries.
  - annotations that provide additional support for query processing and extensibility.
- the Epoq approach to optimizer extensibility (Chapter 5).
  - strategy extensibility — a new kind of extensibility in query optimizers.
  - integration of independent strategies for controlling the optimization process.
  - characterization of the optimizer control problem.
  - a formal characterization of query processing strategies.

- the Epoq architecture (Chapter 6).
  - a definition for a common interface that characterizes processing strategies.
  - description of the requirements for control over optimizer execution.
  - an extensible control based on planning the optimization process.

The Epoq architecture was motivated by the variety of problems that arise in optimizing queries in object-oriented systems, and by the different solutions to some of these problems. An additional contribution of this thesis is the categorization of optimization problems by object-oriented modelling feature (Chapter 4).

We also defined a complete example rewrite system illustrating the architecture and its extensibility (Chapter 8). The join conversion region in this example (region CJ) was designed and implemented as part of the research for this thesis. This region addresses the previously unexplored problem of subqueries nested in the output specification of a query expression.

9.3 Research Directions

There are a number of directions for research to enhance the work presented in this thesis and to explore new applications for the ideas about extensibility presented here.

The EQUAL algebra provides operators to work over sets. It would be interesting to explore extensions to the language to consider more complex bulk types. It would also be interesting to explore the creation of arbitrary new types by the algebraic operations. EQUAL can build objects with new complex types out of parameterized sets and tuples, and other languages build values with new types constructed from their basic types (sets, tuples, lists, arrays) (e.g [32], [150]). We would like to explore building new abstract objects in response to queries. For example, if we define an abstract data type D, we would like to write a query that can build new objects of type D. Such a capability might be also be interesting to explore as a way to define views in o-o database systems.

We incorporated a cost model into our query processing system, but did not actually develop a cost model in this thesis. The development of a cost model for queries over objects that are instances of abstract data types is still an open area of research. A major problem to be addressed is the estimation of the cost of expressions containing arbitrary methods over abstract data types. In the thesis we described how the parameters of such a model could be incorporated into the query representation, with the methods of a cost model defined as methods over the representation. We would like to see the development of a comprehensive cost model for object-oriented database systems. Such a model should include imprecise cost estimates for high-level queries as well as more precise estimates for access plans. The cost model would also have to respond to the extensibility of the system.

A few areas of the architecture were not completely developed in this thesis. In particular, we deferred the question of choosing subqueries or alternative queries for processing by the optimizer. Although the architecture describes the requirement to initially choose a query and goal for processing, store alternative queries, and choose a result query, the implementation in the example made simple choices for all of these. It would be useful to explore more complex implementations for these methods. The planning rules may also offer assistance in determining subqueries and goals to process within the optimizer. For example, we envision rules with actions that can find subqueries of the rule's input query. The rule can either either act on those subqueries immediately, or place the subqueries and goals into global memory to be chosen later by the high- level region execution module.

We have a number of ideas for optimization strategies that could be included as regions in an Epoq optimizer. First of all, we would like to further explore the transformations that can be performed on nested queries. This work would extend the implementation work that was done on region CJ in the example optimizer of Chapter 8. We have also done some preliminary work on the use of cost-based heuristics to guide query transformation. The query representation can provide information about the cost of data and operations. We propose defining rewrite rules that are annotated with cost considerations, and using the cost information to help an optimizer module determine which rules to apply to a query expression. We currently envision this as a leaf region with its own rule search engine, but these ideas may generalize to control over the optimization process.

It would also be interesting to experiment with different optimizer controls. We described a control that results in sequential execution of regions. We believe the architecture would also support parallel control over subordinate region execution.

We intend to develop a complete Epoq optimizer. This requires the development of a number of subordinate regions to manipulate queries. In this thesis we have described regions that rewrite high-level query expressions. We need to also incorporate regions that develop access plans for queries. The development of an optimizer involves not only the development of such regions, but a design for how the regions can be used in the optimizer.

A methodology for designing Epoq optimizers is still an open area of research. Epoq provides the ability to combine many different optimization strategies in different ways depending on the query being processed. Although the architecture describes the mechanism for combining these strategies, good heuristics describing which strategies to combine for which queries still need to be developed. Fortunately, the extensibility of the Epoq architecture provides the mechanism for experimenting with different heuristics. This architecture could also be used as a vehicle for testing and comparing different optimization strategies.

9.4 Final Thoughts

The overall approach to extensibility described here may very well turn out to be the most important contribution of this thesis. In particular, we believe that our approach to extensibility and control in query optimization is more broadly applicable than just to object-oriented query processing. First of all, this approach could be applied to build optimizers for any database model, not just the object-oriented model. It is possible that our hierarchical planning-based control, with central points for integrating independent components, might also be applied to control in distributed database systems. Query optimizers on different machines might be combined, using the Epoq approach, to process a single query.

The Epoq approach to extensibility can also be viewed as an approach to integration. The strategies in an Epoq optimizer can come from any source. The architecture combines strategies built using our planning-based control, strategies built with the knowledge that they will be incorporated into a particular optimizer, and strategies built completely independently of the optimizer that will eventually use them. The ability to incorporate the latter implies that the Epoq approach could be applied to problems in heterogeneous databases.

The ability within Epoq to integrate independent processing strategies also has implications for database engineering. The Epoq approach is an open approach to system development. This approach allows the optimizer for a particular system to be extended with modules built by independent developers. Conversely, a module to address a particular optimization problem can be built independently of the optimizer that will eventually use it. Epoq supports such autonomous

development. We expect the development of query processing systems of the future to take a similar approach.

Appendix A

EQUAL Query Transformations

This appendix contains transformations for EQUAL expressions. This is not a complete set of transformations, nor is it minimal. It is an inventory to illustrate the transformations that can be done with the EQUAL operations. Some of these transformations appeared in [132] and [133] (also Table 3.5). The rest are previously unpublished.

The transformation rules given here illustrate that many rewrite rules for the relational algebra can be generalized to EQUAL structural equivalences. We do not, in general, give rules for transforming Boolean predicate expressions but believe that common Boolean transformations will also generalize to EQUAL.

All of the transformations in this list are structural equivalences. (See Table 3.4 for some examples of weaker equivalences.) As noted in Section 3.3.1, EQUAL always produces new objects as the result of queries. These objects cannot be identical, and will at best be shallow-equal. In other words, no two query expressions will ever produce $\equiv_0$ results, but will produce $\equiv_i$ results for some i $\geq$ 1. In all of the following transformations we use $\equiv$ to mean $\equiv_1$ as the default equivalence.

In addition, the equivalence levels given assume that there are no nested queries, other than those explicitly shown. Queries nested in input, function, or predicate expressions could increase the equivalence depth. Some transformations, when involving nested queries, could also require duplicate elimination in one direction.

NOTATION. We use pluralized capital letters (As, Bs, Ss, etc.) to indicate database sets. Capital letters (A, B, etc.) represent names (in particular, attribute names or property names). Lower case letters (a, b, s, etc.) represent variable names, with the exception that lower case p's (p, p_i, p′, etc.) are used as names for Boolean predicates, and f and g are function names.

TRANSFORMATION A.1. *Set Union.*

Union(As,Bs) $\equiv$ Union(Bs,As)

TRANSFORMATION A.2. *Union associativity.*

Union(Union(As,Bs),Cs) $\equiv$ Union(As,Union(Bs,Cs))

Assumptions:

- A type has only a single immediate subtype. If this is not true then the transformation may not preserve result type.

TRANSFORMATION A.3. *Intersection commutativity.*

$$\text{Intersection(As,Bs)} \quad \equiv \quad \text{Intersection(Bs,As)}$$

TRANSFORMATION A.4. *Collapsing nested Select.*

$$\text{Select(Select(As,}\lambda\text{a p(a)),}\lambda\text{b } p_1\text{(b))} \quad \equiv \quad \text{Select(As,}\lambda\text{a p(a)} \wedge p_1\text{(a))}$$

TRANSFORMATION A.5. *Select ordering.*

$$\text{Select(Select(As,}\lambda\text{a p(a)),}\lambda\text{b } p_1\text{(b))} \quad \equiv \quad \text{Select(Select(As,}\lambda\text{a } p_1\text{(a)),}\lambda\text{b p(b))}$$

TRANSFORMATION A.6. *Unnecessary Select.*

$$\text{Select(As,}\lambda\text{a True)} \quad \equiv \quad \text{As}$$

TRANSFORMATION A.7. *Select Intersection.*

$$\text{Intersection(Select(Ss,}\lambda\text{s } p_1\text{(s)), Select(Ss,}\lambda\text{s } p_2\text{(s)))} \equiv \text{Select(Ss,}\lambda\text{s } p_1\text{(s)} \wedge p_2\text{(s))}$$

TRANSFORMATION A.8. *Select Union.*

$$\text{Union(Select(Ss,}\lambda\text{s } p_1\text{(s)), Select(Ss,}\lambda\text{s } p_2\text{(s))} \quad \equiv \quad \text{Select(Ss,}\lambda\text{s } p_1\text{(s)} \vee p_2\text{(s))}$$

TRANSFORMATION A.9. *Union as Select.*

$$\text{Union(As, Bs)} \quad \equiv \quad \text{Select(Ss,}\lambda\text{s s} \in \text{As} \vee \text{s} \in \text{Bs)}$$

Assumptions:

- Ss is a superset of As and Bs, and the member-type of Ss is the closest common supertype of the member-type of As and Bs.

TRANSFORMATION A.10. *Removing* $\forall$.

$$\text{Select(As,}\lambda\text{a } \forall\text{b(b} \in \text{Bs} \wedge \text{p(a,b)))} \equiv \text{Select(As,}\lambda\text{a Empty(Difference(Bs, Select(Bs,}\lambda\text{b p(a,b)))))}$$

TRANSFORMATION A.11. *Removing* $\exists$.

$$\text{Select(As,}\lambda\text{a } \exists\text{b(b} \in \text{Bs} \wedge \text{p(a,b)))} \equiv \text{Image(Ojoin(As,Bs,A,B,}\lambda\text{a,b p(a,b)),}\lambda\text{t t.A)}$$

Assumptions:

- Bs is independent of a.

TRANSFORMATION A.12. *Difference as negation.*

$$\text{Difference}(\text{As}, \text{Select}(\text{As},\lambda a\ p(a))) \quad \equiv \quad \text{Select}(\text{As},\lambda a\ \neg p(a))$$

TRANSFORMATION A.13. *Path tracing as set membership.*

$$\text{Select}(\text{Ss},\lambda s\ s.A_1.A_2 \sim x) \quad \equiv \quad \text{Select}(\text{Ss},\lambda s\ s.A_1 \in \text{Select}(\text{Rs},\lambda r\ r.A_2 \sim x))$$

Assumptions:

- Rs is (a superset of) the range of A_1 applied to an element of Ss (i.e., $\forall s \in \text{Ss}\ s.A_1 \in \text{Rs}$).
- $\sim$ is a binary Boolean operator defined for the result type of A_2 and the type of x. For example, this transformation is useful when $\sim$ is set membership, the result of A_2 has type T and x has type Set[T].

TRANSFORMATION A.14. *Single-step path as set membership.*

$$\text{Select}(\text{Ss},\lambda s\ s.A \sim x) \quad \equiv \quad \text{Select}(\text{Ss},\lambda s\ s.A \in \text{Select}(\text{Rs},\lambda r\ r \sim x))$$

Assumptions:

- Rs is (a superset of) the range of A applied to an element of Ss.
- $\sim$ is a binary Boolean operator defined for the member-type of Rs (result type of A) and the type of x.

TRANSFORMATION A.15. *Path tracing as join.*

$$\text{Select}(\text{Ss},\lambda s\ s.A_1 \sim x) \equiv \text{Image}(\text{Ojoin}(\text{Ss},\text{Select}(\text{As},\lambda a\ a \sim x),\text{S},\text{A},\lambda s,\text{a}\ s.A_1 =_0 a),\lambda t\ t.A)$$

Assumptions:

- As is (a superset of) the range of A applied to an element of Ss.
- x is independent of $s \in \text{Ss}$.
- $\sim$ is a binary Boolean operator defined over the type of the A property of the member-type of Ss and the type of x.

TRANSFORMATION A.16. *Inverses.*

$$\text{Select}(Ss,\lambda s\ s.A_1 \in Rs) \quad \equiv \quad \text{Image}(Rs,\lambda r\ \text{inverse}(r))$$

Assumptions:

- The A_1 property of Ss has an inverse.
- A_1 is a one-to-one property. (If A_1 is many-to-one then the inverse returns a Set and the Image result will have to be Flattened.)

TRANSFORMATION A.17. *Composition of Images.*

$$\text{Image}(\text{Image}(Ss,\lambda s\ f(s)),\lambda r\ g(r)) \quad \equiv \quad \text{Image}(Ss,\lambda s\ g(f(s)))$$

TRANSFORMATION A.18. *Union of Images.*

$$\text{Union}(\text{Image}(As,\lambda a\ f(a)), \text{Image}(Bs,\lambda b\ f(b))) \quad \equiv \quad \text{Image}(\text{Union}(As, Bs),\lambda s\ f(s))$$

Assumptions:

- The member-type of As is an ancestor or descendant of the member-type of Bs.

Note that the assumption ensures static typing. If the member-types of As and Bs are subtypes of a common supertype different from A or B, then f may not be defined for the supertype. Additionally, even if f is defined, the result of Union(Image...) could generate a member-type that is a subtype of the member-type of the result of Image(Union...).

TRANSFORMATION A.19. *Folding Image into Project.*

$$\begin{aligned}&\text{Project}(\text{Image}(As,\lambda a\ f(a)),\lambda s < \ldots,(A_i,g_i(s)),\ldots >)\\ &\qquad\qquad \equiv_2 \quad \text{Project}(As,\lambda a < \ldots,(A_i,g_i(f(a))),\ldots >)\end{aligned}$$

Assumptions:

- Image is folded into all A_i's such that g_i is a function of s. (If g_i is independent of s it is treated as a constant and is the same on either side of the transformation.)

TRANSFORMATION A.20. *Select through Project.*

$$\begin{aligned}&\text{Project}(\text{Select}(As,\lambda a\ p(a)),\lambda t < \ldots,(A_i,t),\ldots,(A_j,f_j(t)),\ldots >)\\ &\qquad \equiv_2 \quad \text{Select}(\text{Project}(As,\lambda a < \ldots,(A_i,a),\ldots,(A_j,f_j(a)),\ldots >),\lambda t\ p(t.A_i))\end{aligned}$$

Recall that tuples are unordered, thus the notation $(A_j,f_j(t))$ represents all attributes that are computed by the application of a function over an element of As. The key in this transformation is that some attribute is computed by applying the identity function (i.e., (A_i,t) exists for some i).

TRANSFORMATION A.21. *UnNest-Nest removes nested duplicates.*

$$\begin{aligned}
&\text{Nest}(\text{UnNest}(\text{Ss},\text{A}_i),\ \text{A}_i) \quad \equiv_3 \quad \text{DupEliminate}(\text{Rs},=_2)\\
&\qquad \text{where Rs} \quad := \quad \text{Project}(\text{Ss},\lambda \text{t} <(\text{A}_1,\text{t.A}_1),\ldots,(\text{A}_{i-1},\text{t.A}_{i-1}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{A}_i,\text{Find_dups(t)}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{A}_{i+1},\text{t.A}_{i+1}),\ldots,(\text{A}_n,\text{t.A}_n)>)\\
&\text{and Find_dups(t)} \quad := \quad \text{Flatten}(\text{Image}(\text{Select}(\text{Ss},\lambda \text{s s.A}_j =_0 \text{t.A}_j\ \forall \text{j}\neq\text{i}),\lambda \text{q q.A}_i))
\end{aligned}$$

Assumptions:

- Ss has type Set[Tuple[$<(\text{A}_1,\text{T}_1),\ldots,(\text{A}_i,\text{Set}[\text{T}_i]),\ldots,(\text{A}_n,\text{T}_n)>$]]

TRANSFORMATION A.22. *Composing Select and Ojoin.*

$$\begin{aligned}
&\text{Select}(\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)}),\lambda \text{t } \text{p}_s(\text{t.A,t.B}))\\
&\qquad\qquad \equiv_2 \quad \text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)} \wedge \text{p}_s(\text{a,b}))
\end{aligned}$$

TRANSFORMATION A.23. *Performing Select before Ojoin.*

$$\begin{aligned}
&\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a)} \wedge \text{p}'(\text{a,b}))\\
&\qquad\qquad \equiv_2 \quad \text{Ojoin}(\text{Select}(\text{As},\lambda \text{a p(a)}),\text{Bs,A,B},\lambda \text{a,b p}'(\text{a,b}))
\end{aligned}$$

TRANSFORMATION A.24. *Pushing Select past Ojoin.*

$$\begin{aligned}
&\text{Select}(\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)}),\lambda \text{t } \text{p}_1(\text{t.A}) \wedge \text{p}_2(\text{t.A,t.B}))\\
&\qquad\qquad \equiv_2 \quad \text{Ojoin}(\text{Select}(\text{As},\lambda \text{a } \text{p}_1(\text{a})),\text{Bs,A,B},\lambda \text{a,b p(a,b)} \wedge \text{p}_2(\text{t.A,t.B}))
\end{aligned}$$

Note that transformation A.24 can be achieved by composing transformations A.22 and A.23.

TRANSFORMATION A.25. *Generalized Select past Ojoin.*

$$\begin{aligned}
&\text{Select}(\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)}),\lambda \text{t } \text{p}_1(\text{t.A}) \wedge \text{p}_2(\text{t.A,t.B}))\\
&\qquad \equiv_2 \quad \text{Select}(\text{Ojoin}(\text{Select}(\text{As},\lambda \text{a } \text{p}_1(\text{a})),\text{Bs,A,B},\lambda \text{a,b p(a,b)}),\lambda \text{t } \text{p}_2(\text{t.A,t.B}))
\end{aligned}$$

TRANSFORMATION A.26. *Ojoin semantics.*

$$\begin{aligned}
&\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)})\\
&\qquad\qquad \equiv_2 \quad \text{Select}(\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b True}),\lambda \text{t p(t.A,t.B)})
\end{aligned}$$

TRANSFORMATION A.27. *Commutativity of Ojoin.*

$$\text{Ojoin}(\text{As,Bs,A,B},\lambda \text{a,b p(a,b)}) \quad \equiv_2 \quad \text{Ojoin}(\text{Bs,As,B,A},\lambda \text{b,a p(a,b)})$$

TRANSFORMATION A.28. *Ojoin associativity.*

$$\begin{aligned}&\text{Ojoin(Ojoin(As,Bs,A,B,}\lambda\text{a,b } p_a(a) \wedge p_b(b) \wedge p_{ab}(a,b)),Cs,,C,\\ &\quad \lambda\text{t,c } p_c(c) \wedge p_{ac}(t.A,c) \wedge p_{bc}(t.B,c) \wedge p_{abc}(t.A,t.B,c))\\ &\qquad \equiv_2 \quad \text{Ojoin(As, Ojoin(Bs,Cs,B,C,}\lambda\text{b,c } p_b(b) \wedge p_c(c) \wedge p_{bc}(b,c)),A,,\\ &\qquad\qquad \lambda\text{a,t } p_a(a) \wedge p_{ab}(a,t.B) \wedge p_{ac}(a,t.C) \wedge p_{abc}(a,t.B,t.C))\end{aligned}$$

TRANSFORMATION A.29. *Ojoin redundancy (1).*

$$\begin{aligned}&\text{Ojoin(As,Bs,A,B,}\lambda\text{a,b p(a,b))}\\ &\qquad \equiv_2 \quad \text{Select(UnNest(Project(As,}\lambda\text{a <(A,a),(B,Bs)>), B),}\lambda\text{t p(t.A,t.B))}\end{aligned}$$

TRANSFORMATION A.30. *Ojoin redundancy (2).*

$$\begin{aligned}&\text{Ojoin(As,Bs,A,B,}\lambda\text{a,b p(a,b))}\\ &\qquad \equiv_2 \quad \text{UnNest(Select(Project(As,}\lambda\text{a <(A,a), (B,Select(Bs,}\lambda\text{b p(a,b)))>),}\\ &\qquad\qquad\qquad \lambda\text{t NOT Empty(t.B)), B)}\end{aligned}$$

TRANSFORMATION A.31. *Coalesce redundancy.*

$$\begin{aligned}&\text{Coalesce(Ss,}A_k\text{,}=_i\text{)} \quad \equiv_{i+2} \quad \text{Project(Ojoin(Ss,NoDup}A_k\text{,,B,}\lambda\text{a,b a.}A_k =_i \text{ b),}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda\text{t <(}A_1\text{,t.}A_1\text{),...,(}A_{k-1}\text{,t.}A_{k-1}\text{),}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A_k\text{,t.B),}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A_{k+1}\text{,t.}A_{k+1}\text{),...,(}A_n\text{,t.}A_n\text{)>)}\\ &\quad \text{where NoDup}A_k \quad := \quad \text{DupEliminate(Image(Ss,}\lambda\text{s s.}A_k\text{), }=_i\text{)}\end{aligned}$$

Assumptions:

- Ss has type Set[Tuple[(A_1,T_1),...,(A_n,T_n)]]

TRANSFORMATION A.32. *Flatten redundancy.*

$$\text{Flatten(Ss)} \quad \equiv \quad \text{Image(UnNest(Project(Ss,}\lambda\text{s <(A,s)>), A), }\lambda\text{a a.A)}$$

Assumptions:

- Ss has type Set[Set[T]] for some type T.

TRANSFORMATION A.33. *UnNest redundancy.*

$$\begin{aligned}&\text{UnNest(Ss,}A_k\text{)} \quad \equiv_2 \quad \text{DupEliminate(Flatten(Image(Ss,}\lambda\text{s}\\ &\qquad\qquad\qquad\qquad \text{Project(s.}A_k\text{,}\lambda\text{t <(}A_1\text{,s.}A_1\text{),...,(}A_{k-1}\text{,s.}A_{k-1}\text{),}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A_k\text{,t),}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (A_{k+1}\text{,s.}A_{k+1}\text{),...,(}A_n\text{,s.}A_n\text{)>))), }=_1\text{)}\end{aligned}$$

Assumptions:

- Ss has type Set[Tuple[(A_1,T_1),...,(A_k,Set[T_k]),...,(A_n,T_n)]]

TRANSFORMATION A.34. *Nest redundancy.*

$$
\begin{aligned}
\text{Nest}(\text{Ss},\text{A}_k) \quad &\equiv_2 \quad \text{Project}(\text{Sless}\text{A}_k,\lambda\text{t } <(\text{A}_1,\text{t.A}_1),\ldots,(\text{A}_{k-1},\text{t.A}_{k-1}),\\
&\qquad\qquad (\text{A}_k,\text{Image}(\text{MatchSs}(\text{t}),\lambda\text{q q.A}_k)\\
&\qquad\qquad (\text{A}_{k+1},\text{t.A}_{k+1}),\ldots,(\text{A}_n,\text{t.A}_n)>)\\
\text{where Sless}\text{A}_k \quad &:= \quad \text{DupEliminate}(\text{Project}(\text{Ss},\lambda\text{s } <(\text{A}_1,\text{s.A}_1),\ldots,(\text{A}_{k-1},\text{s.A}_{k-1}),\\
&\qquad\qquad\qquad\qquad (\text{A}_{k+1},\text{s.A}_{k+1}),\ldots,\\
&\qquad\qquad\qquad\qquad (\text{A}_n,\text{s.A}_n)>),=_1)\\
\text{and MatchSs}(\text{t}) \quad &:= \quad \text{Select}(\text{Ss},\lambda\text{s } (\text{s.A}_i =_0 \text{t.A}_i\ \forall \text{i}\neq\text{k}))
\end{aligned}
$$

Assumptions:

- Ss has type Set[Tuple[<(A_1,T_1),...,(A_n,T_n)>]]

Bibliography

[1] Serge Abiteboul and Catriel Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical Report No. 846, INRIA, 1988.

[2] Serge Abiteboul and Nicole Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, Dec 1986.

[3] Serge Abiteboul and Richard Hull. IFO: A Formal Semantic Database Model. In *Proceedings of the Symposium on Principles of Database Systems*, pages 119–129. ACM, April 1984.

[4] Serge Abiteboul and Paris Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, 21(3):9–18, Summer 1990.

[5] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In *SIGMOD Proceedings*, pages 159–173. ACM, 1989.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA., 1986.

[7] M. M. Astrahan et al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[8] Malcolm Atkinson et al. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, December 1989.

[9] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object Oriented Database System: the Story of O_2*. Morgan Kaufmann, San Mateo, CA, 1991.

[10] François Bancilhon, Sophie Cluet, and Claude Delobel. A Query Language for the O_2 Object-Oriented Database System. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, pages 122–138, June 1989.

[11] Francois Bancilhon and Setrag Khoshafian. A Calculus for Complex Objects. In *Proceedings of the Symposium on Principles of Database Systems*, pages 53–59. ACM, March 1986.

[12] Jay Banerjee et al. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[13] Jay Banerjee, Won Kim, and Kyung-Chang Kim. Queries in Object-Oriented Databases. In *Proceedings 4th Intl. Conf. on Data Engineering*, pages 31–38. IEEE, Feb 1988.

[14] D. S. Batory et al. GENESIS: An Extensible Database Management System. In Zdonik and Maier [156], pages 500–518.

[15] Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proceedings ICDT*, Paris, France, 1990.

[16] Véronique Benzaken and Claude Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2. In Dearle et al. [41], pages 403–412.

[17] Philip A. Bernstein and Dah-Ming W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, Jan 1981.

[18] Philip A. Bernstein et al. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, Dec 1981.

[19] E. Bertino. A Survey of Indexing Techniques for Object-Oriented Databases. In J.C. Freytag, G. Vossen, and D. Maier, editors, *To appear in Query Processing for Advanced Database Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[20] Elisa Bertino. Method Precomputation in Object-Oriented Databases. *SIGOIS Bulletin*, 12(2,3):199–212, 1991.

[21] Elisa Bertino and Paola Foscoli. An Analytical Model of Object-Oriented Query Costs. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*. Springer-Verlag, 1992.

[22] Elisa Bertino and Won Kim. Indexing Techniques for Queries on Nested Objects. Technical Report ACT-OODS-132-89, MCC, 1989.

[23] Michael Carey and Laura Haas. Extensible Database Management Systems. *SIGMOD Record*, 19(4):54–60, Dec 1990.

[24] Michael J. Carey, David J. DeWitt, et al. The Architecture of the EXODUS Extensible DBMS. In Dittrich and Dayal [44], pages 52–65.

[25] Arvola Chan et al. Storage and Access Structures to Support a Semantic Data Model. In *Proceedings of the 8th VLDB Conference*, pages 122–130. ACM, 1982.

[26] Ashok K. Chandra. Theory of database queries. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–9. ACM, March 1988.

[27] Ellis E. Chang and Randy H. Katz. Exploiting Inheritance and Structure Semantics for Effective clustering and Buffering in an Object-Oriented DBMS. In *SIGMOD Proceedings*, pages 348–357. ACM, 1989.

[28] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, MA., 1985.

[29] Sophie Cluet. *Langages et Optimisation de Requêtes pour Systèmes de Gestion de Base de Données Orientés-Objet*. PhD thesis, Université de Paris-Sud - Centre d'Orsay, June 1991.

[30] Sophie Cluet and Claude Delobel. Towards a Unification of Rewrite Based Optimization Techniques for Object-Oriented Queries. In *Septièmes Journées Base de Données Avancées*, Lyons, France, Sep 1991. This paper is an early version of [31].

[31] Sophie Cluet and Claude Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *SIGMOD Proceedings*, pages 383–392. ACM, June 1992.

[32] Sophie Cluet et al. Reloop, an Algebra Based Query Language for an Object-Oriented Database System. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 294–316, November 1989.

[33] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[34] Thomas A. Cooper and Nancy Wogrin. *Rule-based Programming with OPS5*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[35] P. Dadam et al. A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. In *SIGMOD Proceedings*, pages 356–367. ACM, May 1986.

[36] Scott Daniels et al. Query Optimization in Revelation, an Overview. *IEEE Data Engineering Bulletin*, 14(2):58–62, June 1991.

[37] C. J. Date. *An Introduction to Database Systems*, volume I. Addison-Wesley, Reading, MA., 5th edition, 1990.

[38] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the 13th VLDB Conference*, pages 197–208, 1987.

[39] Umeshwar Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, June 1989.

[40] Thomas L. Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann, San Mateo, CA, 1991.

[41] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases: Principles and Practice*. Fourth International Workshop on Persistent Object Systems, Morgan Kaufmann Publishers, Inc., 1990.

[42] Claude Delobel, May 1989. Personal communication.

[43] K. R. Dittrich, editor. *Advances in Object-Oriented Database Systems*. International Workshop on Object-Oriented Database Systems, Springer-Verlag, September 1988.

[44] Klaus Dittrich and Umeshwar Dayal, editors. *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*. IEEE Computer Society Press, September 1986.

[45] Béatrice Finance and Georges Gardarin. A Rule-Based Query Rewriter in an Extensible DBMS. In *Proceedings of the 7th International Conference on Data Engineering*, pages 248–256. IEEE, 1991.

[46] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, January 1989. YALEU/CSD/RR#672.

[47] Patrick C. Fischer and Dirk Van Gucht. Weak Multivalued Dependencies. In *Proceedings of the Symposium on Principles of Database Systems*, pages 266–274. ACM, April 1984.

[48] Johann Cristoph Freytag. A Rule-Based View of Query Optimization. In *SIGMOD Proceedings*, pages 173–180, May 1987.

[49] César Galindo-Legaria and Renato Barrera. A Rule-Based Query Optimizer. Technical Report 92, Centro de Investigacion y de Estudios Avanzados Del IPN, 1989.

[50] César Galindo-Legaria and Arnon Rosenthal. How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin. In *Proceedings of the 8th International Conference on Data Engineering*, pages 402–409. IEEE, 1992.

[51] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *SIGMOD Proceedings*, pages 23–33, May 1987.

[52] Michael P. Georgeff and Amy L. Lansky. Reactive Reasoning and Planning. In *AIII-87*, pages 677–682. AIII, 1987.

[53] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[54] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, Univ. of Wisconsin-Madison, November 1987.

[55] Goetz Graefe, editor. *Workshop on Database Query Optimization*, Portland, OR, May 1989.

[56] Goetz Graefe. Volcano, an Extensible and Parallel Query Evaluation System. Technical Report CU-CS-481-90, University of Colorado at Boulder, July 1990.

[57] Goetz Graefe. Query Evaluation Techniques for Large Databases. Technical Report CU-CS-579-92, University of Colorado at Boulder, 1992. To appear in ACM Computing Surveys.

[58] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *SIGMOD Proceedings*, pages 160–172. ACM, May 1987.

[59] Goetz Graefe et al. Extensible Query Optimization and Parallel Execution in Volcano. In *Proceedings of the Dagstuhl Query Workshop*, 1992. To appear.

[60] Goetz Graefe and David Maier. Query Optimization in Object-Oriented Database Systems: A Prospectus. In Dittrich [43], pages 358–363.

[61] Goetz Graefe and William McKenna. The Volcano Optimizer Generator. Technical Report Computer Science Technical Report 563, University of Colorado at Boulder, December 1991.

[62] Goetz Graefe and Karen Ward. Dynamic Query Evaluation Plans. In *SIGMOD Proceedings*, pages 358–366. ACM, June 1989.

[63] Dirk Van Gucht and Patrick C. Fischer. Some Classes of Multilevel Relational Structures. In *Proceedings of the Symposium on Principles of Database Systems*, pages 60–69. ACM, March 1986.

[64] Laura M. Haas et al. Extensible Query Processing in Starburst. In *SIGMOD Proceedings*, pages 377–388. ACM, June 1989.

[65] Michael Hammer and Dennis McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[66] Michael Hammer and Stanley B. Zdonik, Jr. Knowledge-based query processing. In *Proceedings of the 6th VLDB Conference*, pages 137–147. ACM, 1980.

[67] Andreas Heuer. Foundations of Relational Object Management Systems. In Dittrich [43], pages 209–212.

[68] Mark F. Hornick and Stanley B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987.

[69] Barron C. Housel and Nan C. Shu. A high-level data manipulation language for hierarchical data structures. In *Proceedings of Conference on Data: Abstraction, Definition and Structure*, pages 155–169. ACM, March 1976.

[70] Richard Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*. Academic Press, 1987.

[71] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *Computing Surveys*, 19(3):201–260, September 1987.

[72] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. In *Proceedings of the 18th VLDB Conference*, pages 103–114, 1992.

[73] G. Jaeschke and H. J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the Symposium on Principles of Database Systems*, pages 124–138. ACM, March 1982.

[74] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[75] B. Paul Jenq et al. Query Processing in Distributed ORION. In *EDBT*, pages 169–187, 1990.

[76] Leslie Pack Kaelbling. Goals as Parallel Program Specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.

[77] Paris Kanellakis, Christophe Lécluse, and Philippe Richard. Introduction to the Data Model. In Bancilhon et al. [9].

[78] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function Materialization in Object Bases. In *SIGMOD Proceedings*, pages 258–267. ACM, 1991.

[79] Alfons Kemper and Guido Moerkotte. Access Support in Object Bases. In *SIGMOD Proceedings*, pages 364–374. ACM, 1990.

[80] Alfons Kemper and Guido Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the 16th VLDB Conference*, pages 290–301, 1990.

[81] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. A Blackboard Architecture for Query Optimization in Object Bases. In *Proceedings of the 19th VLDB Conference*, 1993. To appear.

[82] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416. ACM, September 1986.

[83] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, Sept 1982.

[84] Won Kim. A Model of Queries for Object-Oriented Databases. Technical Report ACA-ST-365-88, MCC, 1988.

[85] Jonathan J. King. QUIST: A System for Semantic Query Optimization in Relational Databases. In *Proceedings of the 7th VLDB Conference*, pages 510–517. ACM, 1981.

[86] Henry F. Korth. Optimization of Object-Retrieval Queries. In Dittrich [43], pages 352–357.

[87] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the 12th VLDB Conference*, pages 128–137, 1986.

[88] Gabriel M. Kuper and Moshe Y. Vardi. A new approach to database logic. In *Proceedings of the Symposium on Principles of Database Systems*, pages 86–96. ACM, April 1984.

[89] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proceedings of the 17th VLDB Conference*, pages 363 – 373, 1991.

[90] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies. In *SIGMOD Proceedings*, pages 256–265. ACM, June 1992.

[91] Rosana S. G. Lanzelotte, Patrick Valduriez, M. Ziane, and J.-P. Cheiney. Optimization of Nonrecursive Queries in OODBs. In *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, December 1991.

[92] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proceedings of the Symposium on Principles of Database Systems*, pages 360–368. ACM, March 1989.

[93] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *SIGMOD Proceedings*, pages 424–433. ACM, June 1988.

[94] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik. The AQUA Data Model and Algebra. Technical Report CS-93-09, Brown University, 1993.

[95] Barbara Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[96] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD Proceedings*, pages 18–27. ACM, June 1988.

[97] David Maier and Jacob Stein. Indexing in an Object-Oriented Database. In Dittrich and Dayal [44], pages 171–182.

[98] David Maier and Jacob Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[99] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 472–482. ACM, September 1986.

[100] Akifumi Makinouchi. A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model. In *Proceedings of the 3rd VLDB Conference*, pages 447–453. ACM, 1977.

[101] Christopher V. Malley and Stanley B. Zdonik. A Knowledge-Based Approach to Query Optimization. In *Proceedings of the First International Conference on Expert Database Systems*, pages 329–344, 1987.

[102] Frank Manola and Umeshwar Dayal. PDM: An Object-Oriented Data Model. In Zdonik and Maier [156].

[103] Bernhard Mirschang. Extending the Relational Algebra to Capture Complex Objects. In *Proceedings of the 15th VLDB Conference*, pages 297–305, 1989.

[104] Gail Mitchell. The Representation of Queries in an OODB. Technical Report (OODB Working Paper 92-01), Brown University, 1992.

[105] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proceedings of the 19th VLDB Conference*, August 1993. To appear.

[106] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik. Optimization of Object-Oriented Queries: Problems and Approaches. In A. Dogac, T. Özsu, A. Biliris, and T. Sellis, editors, *Object-Oriented Databases*. Springer-Verlag, 1993. NATO ASI Series F. To appear.

[107] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. Object-Oriented Query Optimization: What's the Problem? Technical Report CS-91-41, Brown University, 1991.

[108] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. An Architecture for Query Processing in Persistent Object Stores. In *Proceedings of the Hawaii International Conference on System Sciences*, volume II, pages 787–798, January 1992.

[109] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the 18th VLDB Conference*, pages 91–102, 1992.

[110] *The O_2 Programmer's Manual.* O_2 Technology, BP105, Rocquencourt, 78153 Le Chesnay Cedex, France, May 1991.

[111] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query Processing in the ObjectStore Database System. In *SIGMOD Proceedings*, pages 403–412. ACM, 1992.

[112] S. L. Osborn. Identity, Equality and Query Optimization. In Dittrich [43], pages 346–351.

[113] Mark Palmer and Stanley B. Zdonik. Fido: A Cache that Learns to Fetch. In *Proceedings of the 17th VLDB Conference*, 1991. Also available as Brown University Tech Report CS-91-15.

[114] Jan Paredaens and Dirk Van Gucht. Possiblities and Limitations of using Flat Operators in Nested Algebra Expressions. In *Proceedings of the Symposium on Principles of Database Systems*, pages 29–38. ACM, March 1988.

[115] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD Proceedings*, pages 39–48. ACM, June 1992.

[116] Steven P. Reiss. Eris: The Design and Implementation of an Experimental Relational Information System. Unpublished manuscript, December 1982.

[117] Steven P. Reiss. The Efficient Implementation of Flexible Relational Database Systems. Technical Report CS-82-02, Brown University, 1982.

[118] Daniel R. Ries et al. Decompilation and Optimization for ADAPLEX: A Procedural Database Language. Technical Report CCA-82-04, Computer Corporation of America, Sep 1983.

[119] Arnon Rosenthal, Umeshwar Dayal, and David Reiner. Speeding a Query Optimizer: The Pilot Pass Approach. Computer Corp. of America, unpublished note.

[120] Arnon Rosenthal and César Galindo-Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. In *SIGMOD Proceedings*, pages 291–299. ACM, 1990.

[121] Arnon Rosenthal and Paul Helman. Understanding and Extending Transformation-Based Optimizers. *IEEE Database Engineering Bulletin*, 9(4), December 1986.

[122] Arnon Rosenthal and David Reiner. Extending the Algebraic Framework of Query Processing to Handle Outerjoins. In *Proceedings of the 10th VLDB Conference*, pages 334–343. ACM, 1984.

[123] Arnon Rosenthal and David S. Reiner. Querying relational views of networks. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 109–124. Springer-Verlag, 1985.

[124] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[125] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES Data Model. In *Proceedings of the 13th VLDB Conference*, pages 83–96, 1987.

[126] G. Saake et al. Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype. In *Proceedings of the 15th VLDB Conference*, pages 307–316, 1989.

[127] M. Scholl et al. VERSO: A Database Machine Based on Nested Relations. In S. Abiteboul, P. Fisher, and H. Schek, editors, *Nested Relations and Complex Objects*, pages 27–49. Springer-Verlag, 1989.

[128] P. Schwarz et al. Extensibility in the Starburst Database System. In Dittrich and Dayal [44], pages 85–92.

[129] Edward Sciore and John Sieg, Jr. A Modular Query Optimizer Generator. In *Proceedings of the 6th International Conference on Data Engineering*, pages 146–153, 1990.

[130] P. Griffiths Selinger et al. Access Path Selection in a Relational Database Management System. In *SIGMOD Proceedings*, pages 23–34. ACM, 1979.

[131] Karen Shannon and Richard Snodgrass. Semantic Clustering. In Dearle et al. [41], pages 389–402.

[132] Gail M. Shaw and Stanley B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 264–278, December 1989.

[133] Gail M. Shaw and Stanley B. Zdonik. An Object-Oriented Query Algebra. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, pages 103–112, June 1989. Reprinted in *IEEE Data Engineering Bulletin*,12,3, September 1989.

[134] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pages 154–162. IEEE, 1990. An early version of this paper appears as Brown University tech report CS-89-19.

[135] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. A System for Semantic Query Optimization. In *SIGMOD Proceedings*, pages 181–195. ACM, 1987.

[136] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[137] John Connor Sieg, Jr. *Making extensible database technology work.* PhD thesis, Boston University, 1989.

[138] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Communications of the ACM*, 8(10):568–579, Oct 1975.

[139] M. Stonebraker et al. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[140] Michael Stonebraker. Inclusion of New Types in Relational Database Systems. In Michael Stonebraker, editor, *Readings in Database Systems*. Morgan Kaufmann Pub. Inc., 1988.

[141] Michael Stonebraker et al. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3), September 1990.

[142] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *SIGMOD Proceedings*, pages 340–355. ACM, 1986.

[143] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[144] Dave D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, Univ. of Alberta, December 1990.

[145] Dave D. Straube and M. Tamer Özsu. Query Transformation Rules for an Object Algebra. Technical Report CS-89-23, University of Alberta, 1989.

[146] Dave D. Straube and M. Tamer Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[147] Dave D. Straube and M. Tamer Özsu. Type Consistency of Queries in an Object-Oriented Database. In *Proceedings of the Joint OOPSLA/ECOOP Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 224–233. ACM, October 1990.

[148] Peter Tan. Identification and Transformation of Queries with Join Operations in EQUAL. Unpublished comments. Brown University.

[149] Jeffrey D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[150] Scott L. Vandenberg and David J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *SIGMOD Proceedings*, pages 158–167. ACM, June 1991.

[151] Scott L. Vandenburg and David J. DeWitt. An Algebra for Complex Objects with Arrays and Identity. Technical Report TR#918, Univ. of Wisconsin-Madison, March 1990.

[152] Eugene Wong and Karel Youssefi. Decomposition – A Strategy for Query Processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.

[153] S. B. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In Zdonik and Maier [156], pages 1–32.

[154] Stanley Zdonik and Gail Mitchell. ENCORE: An Object-Oriented Approach to Database Modelling and Querying. *IEEE Data Engineering Bulletin*, 14(2):53–57, June 1991.

[155] Stanley B. Zdonik. Query Optimization in Object-Oriented Database Systems. In *Proceedings of the Hawaii International Conference on System Science*, January 1989.

[156] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, CA, 1990.

[157] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.