QUERY OPTIMIZATION FOR FEDERATED DATABASE SYSTEMS: THE CYRANO PROTOTYPE

by

Zhao-Ping Yu

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Csaba Egyhazy

William Frakes

Sanjay Gupta

October, 1996
Falls Church, Virginia

Key Words: FDBS, Deductive, Object-Oriented, Query, Optimization

QUERY OPTIMIZATION FOR FEDERATED DATABASE SYSTEMS: THE CYRANO PROTOTYPE

by

Zhao-Ping Yu

Csaba Egyhazy, Chairman

Computer Science

(ABSTRACT)

The purpose of this research is to improve the performance for the query processing of Cyrano, a prototype deductive object-oriented meta model for Federated Database Systems (FDBSs). The hypothesis was that query optimization techniques such as Semi-Naive algorithm and Magic-Sets Rewrite algorithm could be used to improve the performance of Cyrano prototype query processing. Query optimization has not been used for an FDBS with a deductive object-oriented meta model. Most existing FDBS query optimization techniques are for FDBSs with relational meta models.

This research involves two major stages. The first stage was to investigate the existing query processing methodologies and query optimization techniques for FDBSs, deductive databases, and object-oriented databases. The research analyzed the methodologies and techniques of representative works. Two typical systems, one from the object-oriented database family and the other from the deductive object-oriented database family, were studied and analyzed in detail. The survey showed that there had been no work reported on query optimization for FDBSs

with deductive object-oriented meta models. The analysis showed that the established query optimization techniques for deductive and object-oriented databases could be viable candidates for query optimization in the Cyrano prototype.

The second stage was to develop a new query processing methodology for Cyrano based on the analytical results of the first stage. A new query processing methodology was proposed, and Semi-Naive and Magic-Sets Rewrite algorithms were employed. Experiments showed that the application of the new query processing methodology improved the performance of the Cyrano query processing up to several hundred percent. Furthermore, the new Cyrano query processing methodology is a general methodology for deductive object-oriented data models, and it can well be applied to other FDBSs with deductive object-oriented meta models.

In conclusion, the research proves that the performance of the Cyrano prototype query processing can be significantly improved with query optimization. It also suggests that query optimization will improve the performance of query processing of other FDBSs with deductive object-oriented meta models.

Dedication

To my wife, DeAnn; my son, Jeremy;

and my parents, Song-Wen and Ting-Hui.

Acknowledgment

I would like to give my special thanks to Dr. Egyhazy, my advisor, for his invaluable guidance. He led me into this challenging yet interesting research. He kept me focused and motivated. He cheered for my accomplishments and gave me encouragement and help when there were difficulties. Without him, the completion of this research would not be possible.

Thanks to Dr. Dzikiewicz. His dissertation was the base of this research. His help in explaining and setting up the Cyrano prototype is one of the important factors leading to the success of this research.

Thanks to the committee members, for their insights and comments.

Thanks also to my little boy Jeremy, for his partial understanding of my working many evenings and weekends; and to my wife DeAnn, for her full support and encouragement during the course of my graduate study.

Table of Contents

List of Figures

List of Tables

1. Introduction

The key concepts of this research are: query, query language, query processing, and query optimization.

A query can be a question issued by some user to a database in order to find some answer from the database. An example might be: "Find all students with a GPA of 3.0." Queries can also be used in transactions that may change the value of the data stored in the database. For example, "Raise the wage of Teaching Assistant to $7 an hour." Finally, queries can be used by Database Management Systems (DBMS) for purposes such as verifying access rights and maintaining the integrity of the database.

A query is a language expression and is usually specified in some query language such as the Structured Query Language (SQL). SQL is the query language used in relational databases. Relational databases are based on the relational data model [Codd70, Date91]. Relational databases are the most widely used traditional database systems today. The query "Find all students with a GPA of 3.0" may be expressed in SQL as follows:

```
SELECT Name
    FROM Student
    WHERE GPA = 3.0
```

where Student is the table which contains the student information in the database.

Queries can also be expressed in calculus and algebra. The corresponding calculus and algebra for SQL are relational calculus and relational algebra [Codd72, Date91]. For databases based on newer data models such as deductive or object-oriented models, queries can be expressed in deductive rules, or object calculus and object algebra, respectively.

Query processing, at minimum, consists of query evaluation. Query evaluation is some program which, given a query and a database, produces the answer to the query.

Query processing may also include query optimization. Query optimization usually tries to minimize the response time for a given query. The optimization process may integrate a number of techniques including logical rewriting of the original query, efficient evaluation of the query, and physical manipulation of data access.

Query, query language, query processing, and query optimization will be discussed in detail in later chapters. The following sections introduce the basics of the Federated Database Systems (FDBS) and focus on one architecture, namely, the one used in the Cyrano prototype [DE94, Dzik96]. The purpose of the research is to provide query optimization for the Cyrano FDBS prototype.

Before the introduction of the Cyrano FDBS prototype, several data models will be introduced. The introduction of these data models is necessary because they are often referenced throughout this thesis.

1.1 Data Models

A data model is a mathematical formalism with two parts: a notation for describing data and a set of operations used to manipulate the data [Ullm88]. Each database management system (DBMS) implements at least one data model that allows the user to see information not as raw bits, but in an understandable format. A database whose DBMS implements a data model X is often classified as an X database. For example, a database whose DBMS implements the relational data model will be classified as a relational database.

Traditional data models include *hierarchical*, *network*, and *relational* models. The hierarchical and network data models are the two earliest data models. The basic structure of these two models is the record of atomic data items. Atomic data items are the most primitive data items, e.g., integers and characters. In the hierarchical model, relationships among data items are represented conceptually as trees, in which references from one record to another lead downward from the root to the leaves.

In the network model, on the other hand, records are linked together as a network. Each node is a record and its edges are references to other records.

The relational model is by far the most widely-implemented data model among the three. The mathematical concept underlying the relational model is the *set-theoretic relation*, or simply *relation*. The members of a relation are *tuples*. A tuple contains k-component of atomic data items. It helps to view a relation as a table in which each row is a tuple and each column corresponds to a component. The columns are often given names called *attributes*. A relational schema specifies the structure of the tables and assigns attribute names to the columns. Relational

query languages are based on first-order predicate calculus or its equivalent relational algebra. SQL is the predominant query language for relational databases, and the SQL standards are universally accepted. A comprehensive introduction to SQL can be found in [MS93].

Newer data models include *deductive* models and *object-oriented* models. Deductive models are also referred to as *logic-based* or *rule-based* models. Deductive models were born and evolved from the artificial intelligence community. The following introduces one of the deductive models - *Datalog*. Datalog is a deductive model specifically designed for databases. A comprehensive introduction to Datalog is presented in [CGT90].

The underlying mathematical model of Datalog is essentially the relational model. The organization of data in Datalog is similar to the relational model, where the basic structures are relations and tuples. However, relations in Datalog do not have named attributes, and references to a component of a tuple are by its position among the components of a given *predicate*.

Predicate symbols are used to denote relations and are used for accessing data. There are two types of predicates - *extensional* and *intensional*. An extensional predicate is a predicate whose relation is stored as data in the database. An intensional predicate is a predicate whose relation is defined by logic rules. The part of the database defined by extensional predicates is referred to as the *extensional database* (EDB). The part of the database defined by logic rules is referred to as the *intensional database* (IDB). An example of an extensional predicate is:

parent (Amy, Mary); which states that Mary is a parent of Amy. An example of an intensional predicate is: ancestor (*X*, *Y*):- parent (*X*, *Y*); which states that *Y* is an ancestor of *X* if *Y* is a parent of *X*.

It is assumed that each predicate symbol either denotes an EDB relation or an IDB relation, but not both. Queries are expressed as logic rules, and writing a query is thus equivalent to specifying an intensional predicate. A query asking for the parents of Amy may be expressed as: query (X):- parent (Amy, X).

Object-oriented models are still evolving and there has not been a universally accepted standard for them. However, certain features are widely accepted. A good introduction of these features appears in [Kim91], where Kim defines a core set of features for the object-oriented model. The following briefly introduces these features:

- Object Identity (OID) - Each real-world entity is an object, with which is associated a system-wide unique identifier. By providing the OID, the system can distinguish two objects that look the same.

- Method - An object may have one or more attributes, and one or more methods which operate on the values of the attributes such as read the value of an attribute or change the value of an attribute.

- Encapsulation - The implementation details of the attributes and methods of an object are hidden from the outside world. The only way to access an object is via the public interface of the object.

- Complex objects - The support of nested structures. In [Kim91], this requirement is not explicitly stated but is implied by the support of *classes*. Ullman stressed the importance of this requirement because support for complex objects is an important prerequisite for supporting classes [Ullm88].

- Class - A class is a definition of a structure together with definitions of the methods by which attribute values of that class can be manipulated. All objects which share the same set of attributes and methods may be grouped into a class. A system that supports encapsulation, methods, and complex objects is said to support classes or abstract data types (ADTs).

- Class Hierarchy - The classes in a system form a hierarchy called a class hierarchy. For a class C and a set of lower-level classes {Si} connected to C, the classes in {Si} are subclasses of the class C, and the class C is the superclass of the classes in {Si}. A class in {Si} is a specialization of class C, and the Class C is the generalization of the classes in the set {Si}.

- In the ADT context, subclasses are called subtypes. "Abstract data type - subtype" and "class - subclass" will be used interchangeably in the rest of the thesis.

- Inheritance - All attributes and methods defined for a class C are inherited by all of its subclasses recursively. An instance of a class S is also a logical instance of all superclasses of S.

In an object-oriented model, queries are often expressed as classes, so that writing a query is the same as defining a class. The query "Find all students with a GPA of 3.0" may be expressed as:

```
QueryClass GoodStudent isA Student
    attribute     STRING NAME;
    constraint    GPA = 3.0;
EndClass.
```

Object Database Management Group (ODMG) has defined an Object Query Language (OQL) in its published standard, ODMG-93[Catt95]. ODMG is the leading standards organization for the world of object database management systems. OQL is essentially the SQL SELECT statement with enhancement supporting ADTs and methods invocation. Since OQL is in reality only a subset of SQL (and thus less powerful than SQL), this research will adopt the approach of using classes or object calculus to express queries, in order to have greater expressive power. Object-oriented queries will be discussed in detail in later chapters.

1.2 Federated Database Systems

Database management systems (DBMSs) provide users with the ability to share effectively large amounts of data stored in multiple files. A DBMS shields its users from the details of file management of multiple files and provides users with a unified view of all data in a database. However, as businesses evolve and merge, as computing systems become more powerful and more accessible to individual users, the demand for accessing multiple databases also grows rapidly. Multiple databases are very likely to have different data models, different semantics in data, and different representations of the data. In many respects, the current accessing of multiple databases is not very different from the accessing of multiple files before the development of DBMSs [CWN94].

The needs and benefits of the integration of multiple databases are well understood. However, it is not feasible to integrate these database systems physically, because most existing database systems were not designed to facilitate such integration. In order to support unified access to different databases without physically integrating those databases, the concept of the Federated Database System (FDBS), also referred to as the Heterogeneous Database System (HDBS) or Multidatabase System (MDBS), has evolved.

An FDBS is a collection of cooperating but autonomous component database systems (DBSs). Component DBSs are integrated to various degrees. The software that provides controlled and coordinated manipulation of the component DBSs is called a *federated database management system* (FDBMS) [SL90]. An FDBMS facilitates the inter-operation of multiple databases while preserving their autonomy, and allows users to access data transparently from multiple heterogeneous databases with a single, relatively simple query. Figure 1 shows a generic FDBS architecture [SL90].

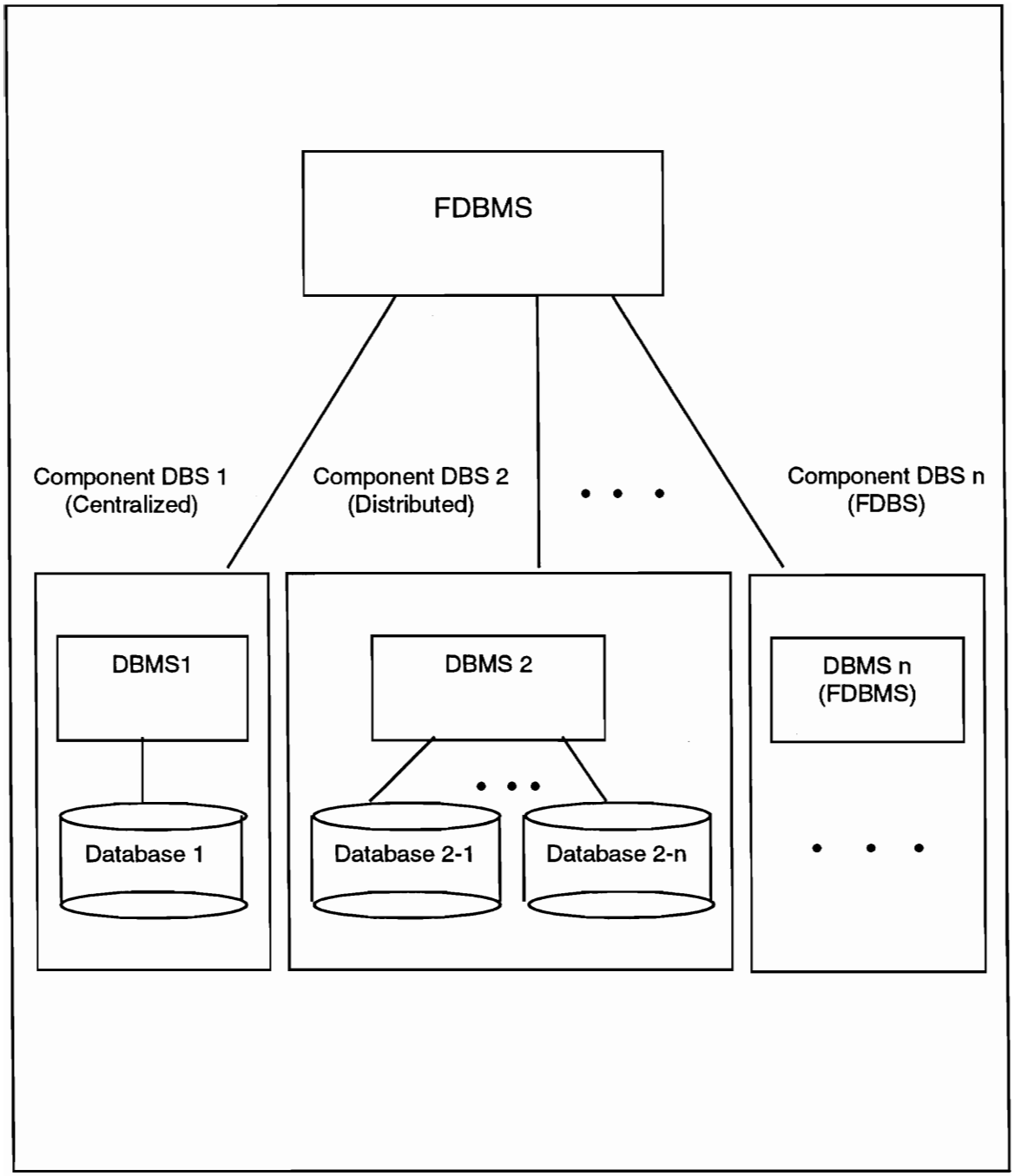

Figure 1 A Federated Database System Architecture

A large body of work has been done on federated database systems. The works are represented in many publications devoted to databases. Examples include [ACM90] and [IEEE91]. In particular, Sheth and Larson provide an excellent introduction to FDBS/MDBS [SL90].

1.3 The Cyrano Federated Database System

The reference architecture supporting the conceptualization of Cyrano is described in [Dzik96, DE94]. This architecture uses the Cyrano deductive object-oriented data model at its core. At the database level it supports the traditional network, hierarchical, and relational databases as well as the newer object-oriented and deductive database systems. Figure 2 shows the Cyrano FDBS reference architecture [Dzik96].

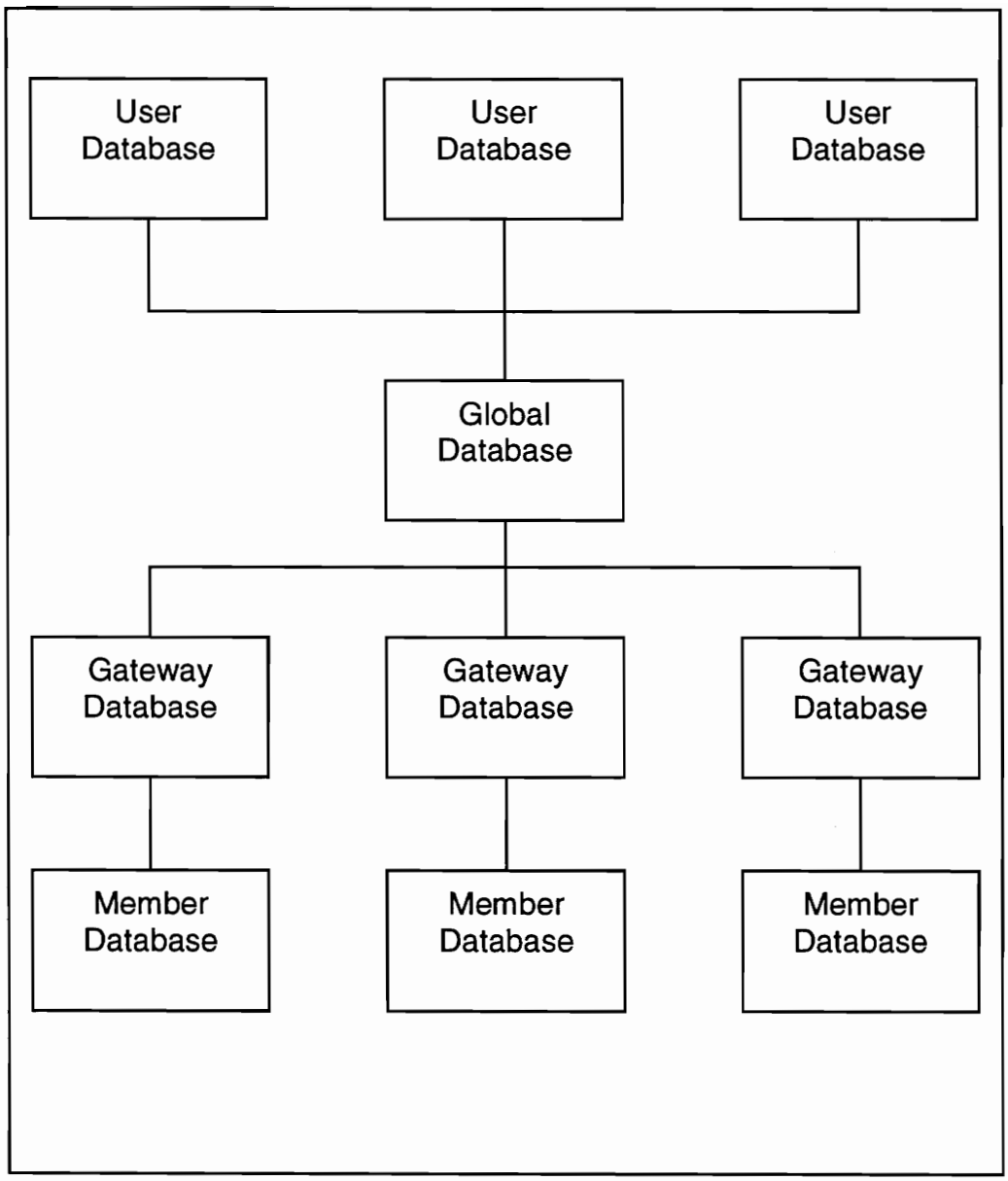

Figure 2 Cyrano Federated Database System Architecture

A *member database* is an autonomous member of the FDBS. It may use any data model supported by the FDBS. It accepts queries in its native language and returns results in its native format.

A *gateway database* is the gateway through which the global database can access the corresponding member database. There is one gateway database per member database. The gateway database translates the queries of the global database into the native language of the member database and translates the answers from the member database language into the global database language.

A *global database* is the federation of one or more gateway databases. It accepts a query in the language of the global database, breaks the query into sub-queries, submits the sub-queries to the appropriate member databases, and merges the results from the member databases to form the result of the original query.

A *user database* is a user view of the global database. The user database may limit a user's access to a subset of the global database and present response data in a form more appropriate to the user.

As can be seen from Figure 2, the Cyrano architecture is similar to that of the generic FDBS. In the Cyrano reference architecture, the global database contains the FDBMS, user databases are the interfaces to the users, and gateway databases are the interfaces to the component (member) databases. What differs between the Cyrano approach and all the other FDBS meta model research is that Cyrano uses deductive rules to resolve the heterogeneity problem among different database systems, while the others emphasize heterogeneity resolution via schemata integration.

1.4 Query Optimization in Federated Database Systems

The essence of query processing is similar in all FDBSs. A generic view of query processing in an FDBS is given in Figure 3.

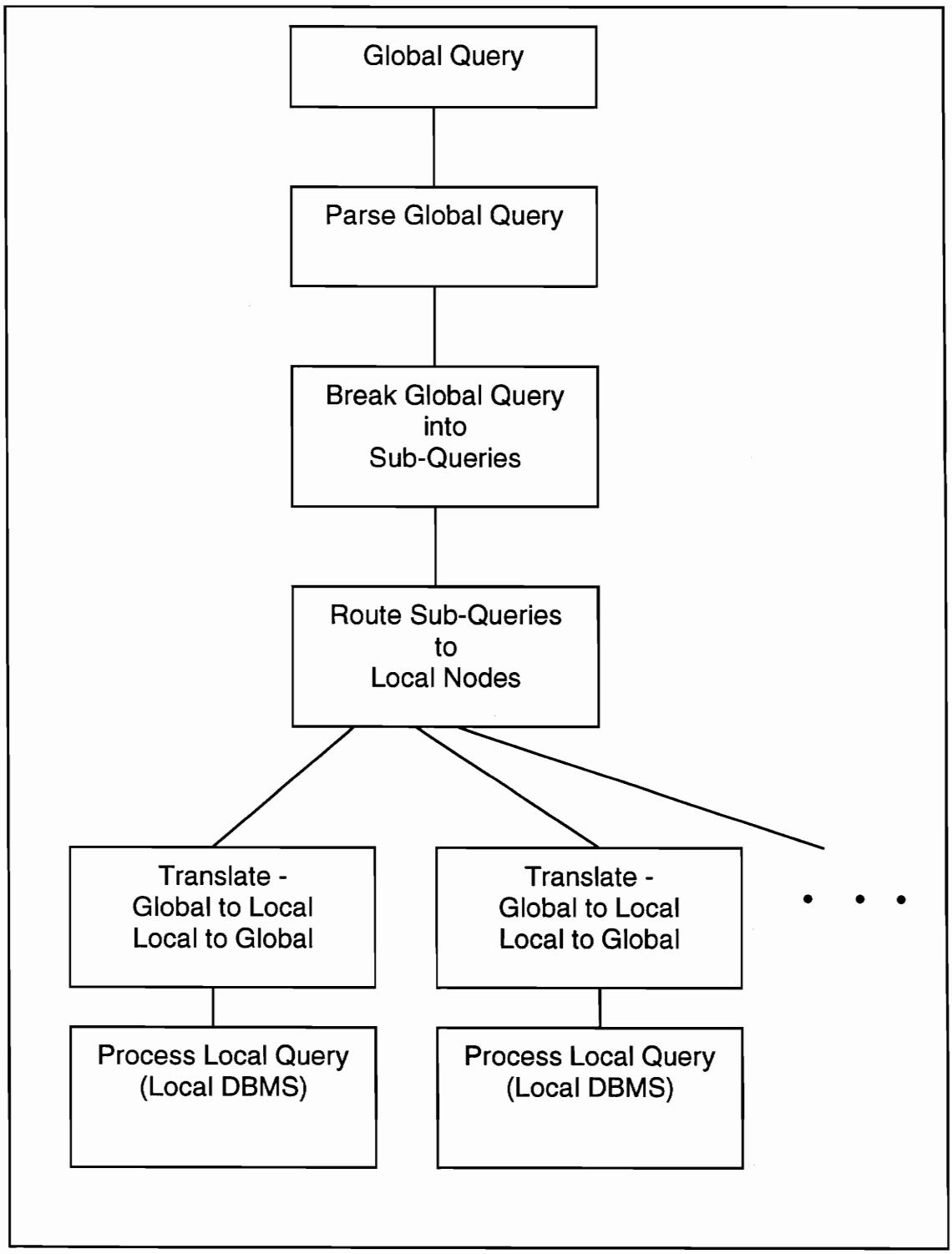

Figure 3 Federated Database System Query Processing

As seen in Figure 3, the global user interface accepts a global query specified in the global query language. The global query is parsed and broken down into sub-queries. The sub-queries are still specified in the global query language. Each sub-query is sent to the corresponding local node. The sub-query is translated into the local query language and sent to the corresponding local DBMS for processing. The response from the local DBMS is then translated back into the global data model format and sent back to the global database. The global database integrates the responses from the local databases into a single global response.

Query optimization in FDBS is difficult mainly due to the autonomy of the local DBMS. It is very difficult, if not impossible, for the global database in an FDBS to obtain all information needed for query optimization. Because of the lack of global control, FDBS query processing can only optimize the path from the global user interface to the local DBMS interface. Actual data access is under the control of the local DBMS query processor.

The distributed nature of FDBSs also complicates query optimization. The global query optimizer must consider communication costs as well, which could involve both the cost of the communication line and the speed of transmission. The global optimizer should so structure global query parsing, decomposition into sub-queries, and the subsequent recombination of sub-query results that the communication costs are minimized.

Finally, global query optimization must deal with the auxiliary information maintained by the global database as well as the local data that is the target of the query. The auxiliary information may include knowledge of replication of data, inter-database relationships, and constraints on inter-database relationships. For exam-

ple, an update transaction on one node may trigger a global transaction to propagate the update to the other sites. Another example might be that if the same data is duplicated in several local databases, a sub-query can be routed to the closest node or the node with best retrieval performance.

As stated by [DKS92] and [LCL95], very few papers have been published on the problems of query processing and query optimization in the context of FDBS. Table 1 [HB96] shows that, of the 50 systems listed, only six systems provide query optimization. All of the six are relational systems and none use a deductive object-oriented meta model as the global model of the FDBS.

Table 1 Federated Database Systems

System Name Organization	Status	Global Data Model	Global Query Optimization
ADDS Amoco Research Center	Prototype	Extended Relational	No
ADMS U. of Maryland	Prototype	Relational	No
AIMS Consort of Datamot, Italy	Prototype	Termino-Logical	No
BLOOM U. of Politecnica de Catalunya, Spain	Research	Object-Oriented	No
Calida GTE Research Lab.	Prototype	Relational	Yes
CIS/TK M.I.T.	Prototype	Relational	Yes
COSMOS-COAX E.T.H. Zurich, Switzerland	Prototype	Object-Oriented	No
DATAPLEX G.E. Research	Prototype	Relational	No

Table 1 Federated Database Systems

System Name Organization	Status	Global Data Model	Global Query Optimization
Davchev U. of Skopje, Yugoslav. Rep. of Macedonia	Research	Relational	No
DAVID NASA	Prototype	Extended Relational	Yes
DOM GTE Lab.	Research	Active Objects	No
DQS and NDMS CRAI, Italy	Prototype	Relational	No
EDDS U of Ulster	Prototype	Relational	No
Empress Rhodius Inc.	Commercial	Relational	No
FEMUS E. Polytech. Fedrale Lausanne Switzerland	Research	Entity-Relationship with Complex Objects	No
FINDIT U. of Colorado	Research	Object-Oriented	No
HD-DBMS UCLA	Research	Entity-relationship	No
Heimbigner U. of Colorado	Prototype	Object-Oriented	No
Hetro NCR Teradata, California	Prototype	Extended Relational	No
HYDRO North Dakota State U.	Research	Relational	No
HSS G.E.	Prototype	Entity Relationship	No
IMDAS NIST	Research	Relational	No
Ingress/Star Relational Tech., Inc.	Commercial	Relational	No
INTENT GMD, Germany	Research	Object-Oriented	No

Table 1 Federated Database Systems

System Name Organization	Status	Global Data Model	Global Query Optimization
InterBase Purdue U.	Prototype	Relational	No
InterViso Dtat Integration, Inc.	Commercial	Relational	Yes
IRO-DB Consor. of EDS, France	Research	Object-Oriented	No
JDDBS Japan Inf. Proc. Dev. Center	Prototype	Relational	No
Jupiter Dublin City U. and Trinity College, Dubin, Ireland	Prototype	Object-Oriented	No
KODIM GMD-IPSI, Germany	Prototype	Object-Oriented	No
LINDA Technical Research Center, Finland	Prototype	Relational	No
MDAS Concordia U., Canada	Prototype	Relational	No
M(DM) IBM	Research	Second Order Logic	No
MRDSM INRIA, France	Prototype	Relational	No
Multi-Star Consor. of CRAI, Italy	Prototype	Relational	No
Multibase Comp. Co. of America	Prototype	Functional	No
Odu U. of Wales	Prototype	Entity-Relationship	No
OIS CNUCE, Italy	Prototype	Object-Oriented	No
OMNIBASE U. of Houston	Prototype	Relational	No
Pegasus HP Labs	Research	Object-Oriented	No

Table 1 Federated Database Systems

System Name Organization	Status	Global Data Model	Global Query Optimization
PRECI U. of Aberdeen	Prototype	Relational	No
Proteus British U.	Prototype	Conceptual	No
SCOOP U. of Paris and Turin	Research	Entity-Relationship	No
SIRIUS-DELTA INRIA, France	Prototype	Relational	No
SIS Athens Faculty of Agri., Greece	Prototype	Relational	Yes
SWIFT SWIFT, Europe	Research	Relational	No
UNIBASE Inst. of Sci., Tech., and Economic Info., Poland	Research	Relational	No
VIP-MDBS Vienna Tech. U.	Prototype	Relational	Yes
XNDM Nat'l Bureau of Standards	Prototype	Relational	No
ZOOIFI U. Zurich, Switzerland	Prototype	Object-Oriented	No

1.5 Approach of the Research

Because of the lack of the existing FDBS query optimization techniques, especially in FDBSs supporting a deductive object-oriented global model, the research begins by reviewing the existing query optimization techniques for deductive databases and object-oriented databases. The main conclusion is that these techniques can well be used for Cyrano FDBS query optimization, and that the per-

formance of the Cyrano FDBS can be improved via the implementation of these techniques. The research supports this claim by implementing one of the efficient evaluation algorithms for the Cyrano FDBS prototype. The implementation significantly reduced the query processing time in the Cyrano FDBS prototype.

1.6 Scope of the Research

This research is an extension to the research initiated by [Dzik96]. The focus here is on improving the performance of query processing in the Cyrano prototype. The improvements will be provided only on the path from the global user interface to the local DBMS interface. The reasons for doing so are twofold. First, the Cyrano prototype was implemented without any distributed member databases. Therefore, the distributed optimization issues are outside the scope of this research. Second, due to the lack of central control over the member databases, the only possible path to be optimized is the path from the global user interface to the local DBMS interfaces, as stated in Section1.4.

This thesis concentrates only on the logical transformation and efficient evaluation aspects of the query optimization. The physical access plan generation may be mentioned when necessary but will not be detailed.

The thesis consists of two main parts: surveys of the existing query processing methodologies and query optimization techniques; and design and development of a new Cyrano query processor.

The survey part involves the investigation of the existing query processing methodologies and query optimization techniques for FDBSs, deductive databases, and object-oriented databases. Since there has been no work reported on que-

ry optimization for FDBSs with deductive object-oriented meta models, and Cyrano prototype is an FDBS with a deductive object-oriented meta model, the thesis presents and analyzes the representative query processing methodologies and query optimization techniques from object-oriented and deductive object-oriented database fields. Challenges for object-oriented query optimization are investigated. In additional to the discussions of the representative works, two typical systems, one from the object-oriented database family and the other from the deductive object-oriented database family, are studied and analyzed in detail. Also, the query processing methodology of the original Cyrano is thoroughly investigated.

The second part is the design and development of a new query processor for the Cyrano prototype. It includes the selection of query processing methodologies and query optimization techniques based on the results of the survey and analytical work; design of a new Cyrano query processing methodology; development of a new Cyrano query processor based on the new methodology; and carrying out the performance comparisons between the original Cyrano and the new Cyrano. Conclusions are drawn from the results of the testing, and further improvement areas are suggested.

1.7 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses queries and query languages. Different views of queries and query language characteristics are provided.

Chapter 3 discusses query processing and query optimization. The chapter introduces query processing methodologies and query optimization requirements.

Chapter 4 presents Cyrano query processing. The chapter details Cyrano query language and Cyrano query processing methodology.

Chapter 5 is the survey of query optimization for object-oriented data models. Challenges for object-oriented query optimization and possible solutions are discussed. Some representative works are presented.

Chapter 6 is the survey of query optimization for deductive object-oriented data models. Evaluation and optimization techniques for deductive data models are provided, and representative works are presented.

Chapter 7 selects query optimization techniques for the Cyano prototype. Different techniques are compared, query optimization for Cyrano is presented, and the implementation status is reported.

Chapter 8 concludes the thesis. Contributions of this research are presented, and future research directions are given.

2. Queries and Query Languages

A query is a high-level specification for a set of objects of interest in a database. It is usually written in a language that specifies what must be retrieved without requiring a statement of how to do it. Query languages are the primary means for defining the declarative interface of a database system [Zdon94, Beer94].

2.1 Three Views of Queries

The basic understanding of what constitutes a query strongly influences the design of a query language and therefore the design and implementation of a query processor. In general, a query language should be declarative and efficient. To be declarative means that users only need to specify what should be retrieved without specifying how the retrieval should be done. To be efficient implies the ease of query optimization which shortens the response time to a query.

Relational query languages have achieved the above requirements. Therefore, advanced query languages such as those designed for object-oriented database access must at least meet these requirements. For this purpose, Staudt et al. proposed an amalgamation of paradigms from object-oriented databases (query-by-class), deductive databases (query-by-rule), and knowledge representation languages (query-by-concept)[SJJN93].

Query-by-class is the preferred approach in object-oriented systems. Object-oriented databases (OODB) combine the paradigms of object identity, class hierarchies, inheritance, methods, and encapsulation from object-oriented programming languages with the functionalities of persistency, concurrency, security, and declarative querying typical in DBMSs.

Query-by-class offers the same object-oriented structures for both questions and answers. A general advantage of representing queries as classes is that answers can be managed the same way as the objects in the OODB. In other words, methods for processing answer objects can be included in query classes. The answer objects can be organized in generalization / aggregation hierarchies, and thereby can reuse the methods of OODB classes. Also, the integration of methods in OODB suggests an implementation approach to query evaluation based on constraints attached to specific subclasses of a query class.

Query-by-rule is the deductive approach. A deductive database consists of a finite set of facts (extensional database), and a finite set of rules (intensional database). Queries in deductive databases look rather simple compared to object-oriented databases. One advantage of query-by-rule is that the membership condition is defined by the rules which have a matching conclusion literal. Thus, it is very easy to formulate a parameterized query by replacing a rule variable by a constant.

Another advantage is that the methods for evaluation / optimization of deductive queries have been thoroughly investigated. Therefore, the implementation could be based on existing techniques.

Query-by-concept is a type inference approach. Concept languages in artificial intelligence have pursued the idea of defining knowledge bases as *type lattices* of so-called *concepts*. A concept is comprised of axioms of the form: $C \subseteq E$ (necessary condition) or $C = E$ (necessary and sufficient condition), where E is constructed from other defined concepts and binary predicates expressing relationships between concepts.

The main purpose of this approach is to relate concepts to each other, i.e., to derive the subsumption between two concepts, C and D. When using this approach, a query is just considered as a new concept which is positioned in the concept lattice by subsumption algorithms. Answers to the query are all instances of all subconcepts and some instances of the direct super concepts which satisfy the additional constraints of the new concept.

The advantage of using the *concept lattices* is that the search space of a query (a new concept) is greatly reduced. The exact placement of a query into the lattice enables intensional query answering, i.e., instead of enumerating all instances of the query, the system answers with the names of the subconcepts of the query.

The query-by-concept approach remains conceptual and has not yet been used, even by its proposers [JGJS95].

2.2 Two Types of Queries

There are two basic types of queries: *recursive* and *non-recursive*. A recursive query is a query which involves some recursive rules. A recursive rule is a rule in which the right side of the rule refers to the left side of the rule, for example:

```
q (x):- ancestor (x, y)
ancestor (x, y):- parent (x, y)
ancestor (x, y):- ancestor (x, z), parent (z, y)
```

where the second *ancestor* rule is a recursive rule (because *ancestor* appears on both sides of the rule), and the query q (x) is therefore a recursive query. Recursive queries are commonly used in querying deductive databases.

Recursive rules can further be classified as *linear recursive rules* or *non-linear recursive rules*. A linear recursive rule is a recursive rule where the right side of the rule refers to the left side of the rule only once. For example, in the second rule of the *ancestor*, the right side of the rule refers only once to *ancestor*.

A non-linear recursive rule is a recursive rule where the right side of the rule refers to the left side of the rule more than once. For example:

ancestor (x, y):- ancestor (x, z), ancestor (z, y)

where the right side of the rule refers to the left side of the rule twice. Since the most commonly used recursive rules are linear recursive rules [BR89, CGT90], this research will consider only the linear recursive rules.

A non-recursive query is a query which does not involve any recursive rules. Non-recursive queries are commonly used in querying relational databases, object-oriented databases, and deductive databases.

2.3 Query Language Characteristics

There are many issues to consider in designing a query language. The following discusses a few key characteristics which are not only important to the query language itself but also have major impact on query optimization. The discussion is based on [SO90] and [Ozsu91]. The key characteristics are:

- formal vs. ad hoc;
- predicates based on structure vs. predicates based on behavior; and
- object-preserving vs. object-generating.

2.3.1 Formal versus Ad Hoc

Formal query languages have several characteristics not found in ad hoc query languages. Most importantly, their semantics are well defined, which simplifies formal proofs about their properties. Common types of formal query languages are a calculus or an algebra.

A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in a calculus may be simplified via certain transformation rules. Ozsu [Ozsu91] argues that a definition of a formal object calculus is needed if declarative languages are to be provided at the user interface. The definition of a calculus requires the resolution of several key issues. The first is the completeness of the calculus. Completeness requires the calculus and the algebra to be equivalent. The second is the safety of the calculus expression. Safe expressions guarantee that queries retrieve a finite set of objects in a finite amount of time. Finally there is a need to develope efficient algorithms to translate safe calculus expressions into algebraic expressions.

Queries expressed in an algebra are procedural in nature. Algebras provide a sound foundation for experimentation with various optimization strategies. Also, a large body of work exists on algebra optimization for other data models, especially for the relational model. Defining a formal object query language in terms of algebra facilitates the comparisons between the object query language and the query languages for the other data models.

2.3.2 Predicates Based on Structure versus Based on Behavior

In terms of structure-based or behavior-based predicates, some languages implement complex objects whose internal structures are visible. Predicates of these languages are structure-based. Other languages view objects as instances of an abstract data type (ADT). Access to objects that are instances of an ADT is accomplished through public interface methods. Methods associated with an object define the behavior of the object. Predicates of these languages are thus behavior-based.

The approach of predicates based on structure violates the encapsulation property of the object-oriented paradigm. But it has the advantage of providing easier cost estimation for the query optimization process [Zdon94].

The approach of predicates based on behavior implements the encapsulation property of the object-oriented paradigm, but it makes the cost estimation difficult, if not impossible. However, Straube and Ozsu argued that the behavior-based approach could include a *get* method for each component of the internal structure of a complex object. Thus a query language supporting predicates based on behavior is more general. It not only preserves the important encapsulation property of the object-oriented paradigm, but it also allows object representations to be introduced for effective query optimization.

2.3.3 Object-Preserving versus Object-Creating

Object-oriented query languages can also be classified as either object-preserving or object-creating. Object-preserving query languages return objects that exist in the original database. Object-creating languages answer queries by creat-

ing new objects from existing objects. A created object should have a unique object identity. There should be some criteria for placing created objects into the inheritance lattice. In one sense this violates the integrity afforded by objects with identity, since objects with no apparent relation to each other can be combined and presented as a new object that encapsulates some well defined behavior. But the requirement for combining existing objects into new objects does exist. One typical example is that in knowledge bases where new knowledge is acquired by forming new facts from existing facts. Also, new objects may need to be created either for output purposes or for further processing.

2.4 Sample Query Languages

This section presents several query languages for object-oriented data models and for deductive object-oriented data models.

2.4.1 Object-Oriented Query Languages

Straube and Ozsu proposed an object calculus and an object algebra [SO90, Stra91] as object-oriented query languages. The languages are formal, behavior-based, and object-preserving query languages. The languages take the object-oriented approach and thus view queries as classes. Among the many proposals of object-oriented query languages and object-oriented query processes investigated by this research, the proposal presented by Straube and Ozsu is the only one that provides both object calculus and object algebra, and the proposal contains one of the most comprehensive query processing methodologies. The Straube and Ozsu query languages and query processing will be discussed in detail in Section 5.3.3.

There have also been several other proposals for object algebra such as those presented by Beeri and Kornatzky (BK-Algebra) [BK93] and by Alhajj and Arkun (AA-Algebra) [AA94].

BK-Algebra is a formal, behavior-based, and object-preserving query language. BK-Algebra is an FP-like language. FP is a functional language [Back78] in which the only way to construct new functions is through a small and fixed set of functional forms, used to produce the new functions from given ones. By selecting functional forms to correspond to common database processing abstractions while allowing an open-ended collection of primitive functions for atomic and composite types, user defined methods can be incorporated easily into the language. The algebra thus achieved as an extensible yet structured database language. The language provides an abstract definition of the collection *constructor*. This constructor provides a general approach to optimize processing over any kind of bulk data, including nontraditional ones. The algebra treats methods as virtual (parameterized) attributes of the types on which they are defined. Methods in the algebra are therefore like parameterized selectors for the objects. Thus algebraic expressions involve methods that can be optimized without special treatment for the methods. The language is also one of the few object-oriented query languages which deal with user-defined methods. However, the methods in the language are for retrieval operations only. One of the other object-oriented query languages which deal with user-defined methods is [SO90].

AA-Algebra views queries as classes. The algebra is a formal, behavior-based, object-creating query language. Alhajj and Arkun claim that most of the existing object-oriented query languages are devoted to the manipulation of existing objects. AA-Algebra supports encapsulation by deriving a set of message expres-

sions to handle the set of objects in a class. That is, the set of total instances of a class or the output from a query is heterogeneous in general, and the only values reachable inside those objects are specified by the corresponding set of message expressions. Thus the algebra deals with heterogeneous sets rather than being restricted to homogeneous sets. The algebra handles stored as well as derived values and hence satisfies computational completeness. The algebra allows for set based predicates and quantifiers, as well as linear recursion. Multiple inheritances are supported where classes are arranged in a lattice. This way, the definition of a class includes the union of all its superclasses.

2.4.2 Deductive Object-Oriented Query Languages

For deductive object-oriented models, Bal and Balsters proposed a language called DTL (DataTypeLog) [BB93]. The language views queries as classes and rules. The language is formal, behavior-based, and object-creating. DTL can be seen as an extension of Datalog, equipped with object identities, complex objects, and multiple inheritance. The language also incorporates the very general notion of sets as first-class objects, i.e., sets are actual terms in the language. DTL defines a simple yet powerful way of combining multiple inheritances with predicates in logic programs. DTL has been proven sound and complete. The language was designed to be used to query object-oriented databases specified with the TM data model [BBV90]. Query optimization was not provided in [BB93]. The possibility of implementing DTL was under investigation during the time of writing [BB93].

Jeusfeld and Staudt presented a deductive object-oriented query language, O-Telos. O-Telos queries can be represented by both classes and rules because O-Telos views queries as both classes and deductive rules. O-Telos is a formal,

behavior-based, and object-creating query language. O-Telos has been implemented in a prototype database, ConceptBase[JS94]. O-Telos will be discussed in detail in Section 6.2.2.

3. Query Processing and Query Optimization

Query processing takes a query as an input, evaluates the query, and gives answers to the query, if any. Query processing may include query optimization. The following subsections describe possible query processing methodologies that can be used in object-oriented database management systems (OODBMSs) and deductive object-oriented database management systems (DOODBMSs) [SO90, CGT90].

3.1 Query Processing for Object-Oriented Databases

Many existing approaches to object-oriented query processing use established relational query processing methodologies with object-oriented extensions. One reason is that OODBMSs have to show that they can be successful, which means that as a minimum they have to provide all the functionalities found in relational database management systems. The other reason is that there is a large body of knowledge and experience in relational query optimization. However, relational query optimization usually does not support the recursion or quantifiers commonly found in deductive systems. Figure 4 depicts the query processing methodology for object-oriented data models proposed by [SO90]. A close examination reveals that the methodology is a typical relational query processing methodology with type checking added.

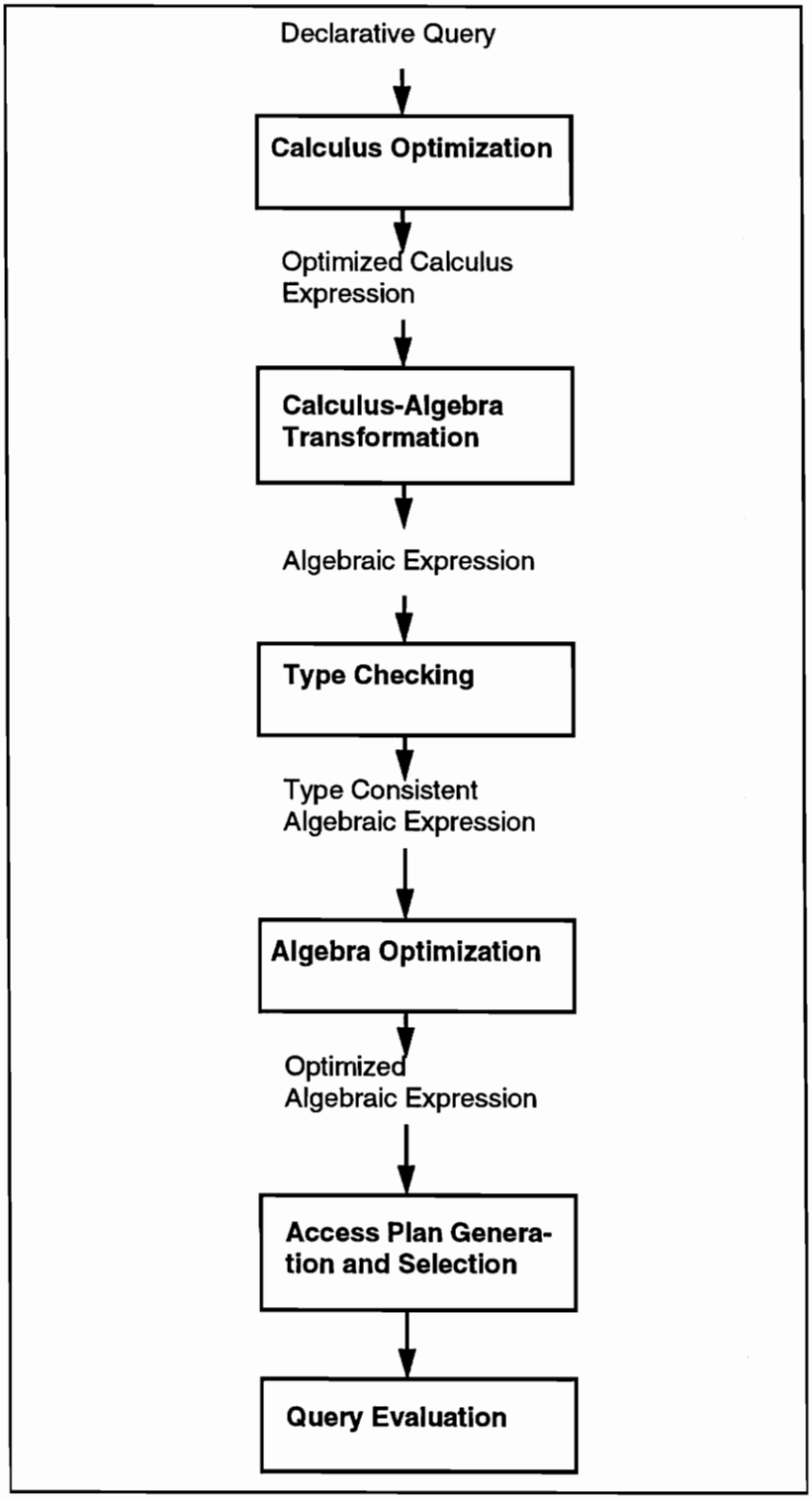

Figure 4 Query Processing for Object-Oriented Data Models

The steps of the methodology are as follows. Queries are expressed in a declarative language such as object calculus. The calculus expression is first optimized to a normal form by eliminating duplicates, applying identities, and rewriting. The normalized calculus expression is then converted to an equivalent object algebraic expression.

The algebraic form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the instances of types in the database. The algebraic expression is then checked for type consistency to ensure that the predicates and methods are not applied to objects which do not support the requested function. The next step is to apply equivalence preserving rewrite rules to the type-consistent algebraic expression. This will usually generate multiple equivalent algebraic expressions.

Access plans are then generated from the algebraic expressions. One of the plans is selected based on certain cost criteria, and the query is evaluated.

3.2 Query Processing for Deductive Object-Oriented Databases

Deductive object-oriented databases attempt to combine the advantages of deductive databases with those of object-oriented models. Most existing approaches to deductive object-oriented query processing include an existing deductive query processing methodology with object-oriented extensions. The main reason for using deductive query processing methodologies with extensions is that many deductive query optimization techniques can be applied directly to the deductive object-oriented model. Figure 5 shows four typical query processing methodolo-

gies for the deductive object-oriented model. The methodologies can be classified as following either a logical approach, an algebraic approach, or a mixed logical-algebraic approach.

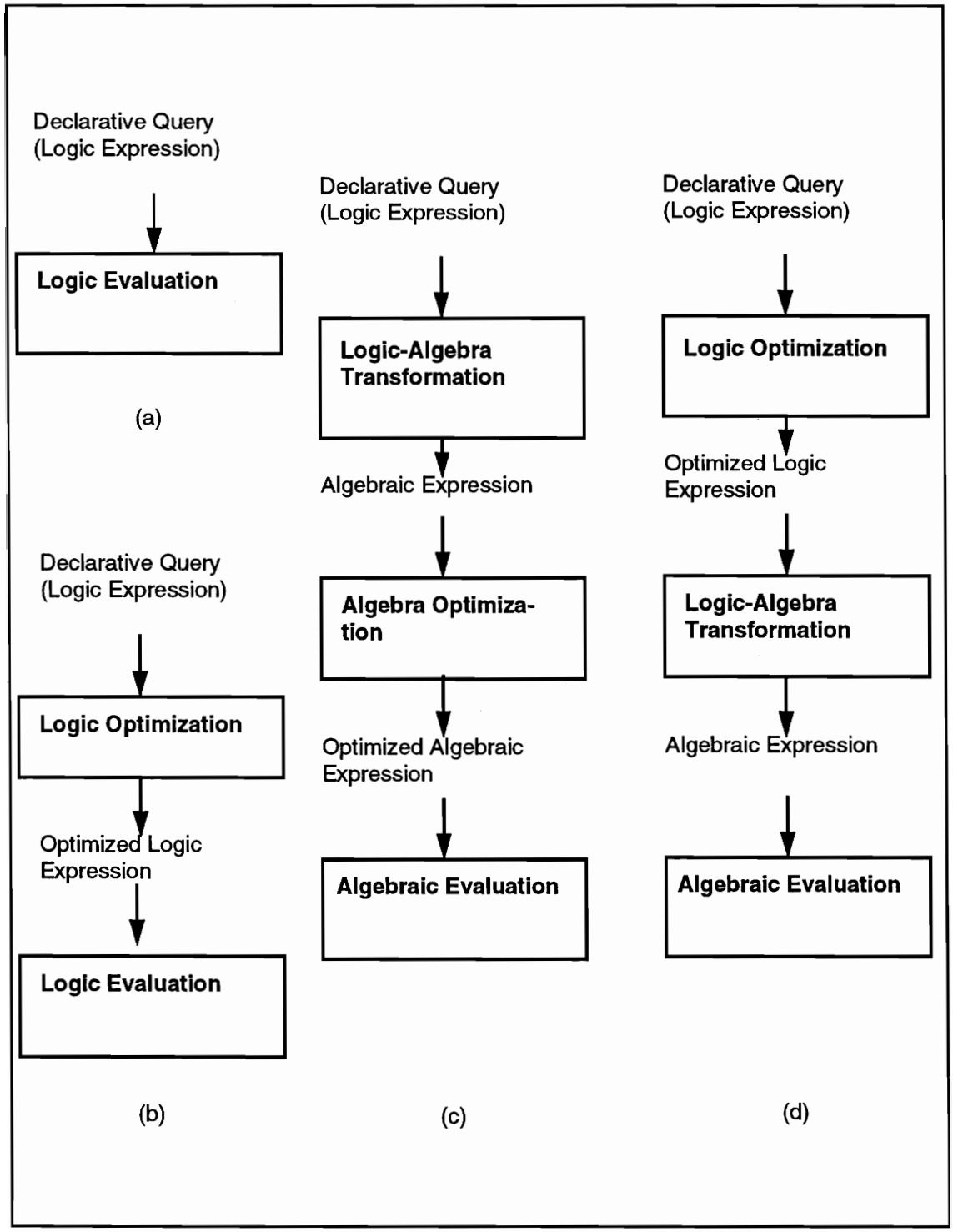

Figure 5 Query Processing for Deductive Data Models

Figure 5 (a) and Figure 5 (b) illustrate the logical approach. Figure 5 (a) shows the simplest query processing in which a logical query is directly evaluated by a logical evaluation method such as the logical version of the *Naive* or *Semi-Naive* method. Figure 5 (b) shows the use of logic rewrite methods and logic evaluation methods in sequence. For example, the *Magic-Sets* method can be used for logic rewrite and the logical version of the *Naive* or *Semi-Naive* method can be used as the logic evaluation method [CGT90].

Figure 5 (c) illustrates the algebraic approach. This approach (1) translates the initial query into an equivalent algebraic expression, (2) optimizes the algebraic query by using an algebraic rewriting method, and (3) performs an algebraic evaluation. In a pure deductive model, there is no type consistency issue during the translation from the logic query to the algebraic query. In a deductive object-oriented data model, however, the type consistency problem has to be resolved during the translation. There are two basic approaches to this problem. The first approach is to use type axioms to guarantee type consistency [JS94]. The second approach is to develop a typed algebra [BB93]. Type consistency issues will be discussed further when some of the deductive object-oriented systems are presented.

Lastly, Figure 5 (d) illustrates the mixed approach. This approach (1) uses a logical rewrite method to optimize the logic expressions, (2) translates the optimized logic expressions into algebraic expressions, and (3) performs an algebraic evaluation that utilizes a method such as the algebraic version of the *Naive* or *Semi-Naive* method [CGT90].

3.3 Requirements for Query Evaluation and Optimization Methods

In order to produce the correct answer in a finite time period, each evaluation and optimization method should satisfy the following requirements:

- A method must be *sound*. In other words, the response produced by the method should not include any object which is not an answer to the query.
- A method must be *complete*. In other words, it should produce all objects which are the answers to the query.
- A method must *terminate*. In other words, the computation should be performed in a finite time interval.

4. Cyrano Prototype Query Language and Query Processing

4.1 Data Model

The Cyrano FDBS global data model is an object-oriented data model as well as a deductive data model. The data model is object-oriented in that it supports the core object-oriented features defined in [Kim91] such as object identity(OID), methods, encapsulation, classes, class hierarchy, and class inheritance. However, OIDs are not implemented in the Cyrano prototype. Instead, addresses are used to access the objects.

Encapsulation in Cyrano is achieved by hiding the implementation details of the attributes and methods. Encapsulation is strictly enforced. Every attribute of a Cyrano class is private. The only way to access an attribute of a Cyrano class is via messages to the public interfaces defined for that class. The reason for doing so is due to the role that Cyrano plays within an FDBS. For example, any access to any data item will require some translation between Cyrano and its member databases. In order to provide unified and transparent access to the users of the FDBS, the implementation details of the translation have to be hidden. Therefore, direct access to any attribute of a Cyrano class is forbidden.

The Cyrano data model is also a deductive data model. This allows Cyrano to create new objects from existing objects based on deductive rules. Cyrano uses deductive rules specified for a class to classify memberships for the class. The deductive approach also allows both quantified and recursive queries, which are usu-

ally not supported by object-oriented models. The most important usage of the deductive rules in Cyrano is to resolve heterogeneity problems among different member databases.

Cyrano defines three system level classes: built-in, gateway, and derived. All other classes, including user defined classes, are defined in terms of these three classes. The BNF of Cyrano classes is given in Appendix A. These Cyrano classes are closely related to Cyrano queries and are presented next.

4.1.1 Built-in Classes

Built-in classes are atomic data types. Currently the Cyrano prototype supports INTEGER, BOOLEAN, and STRING types. The Cyrano definition of a built-in class includes the operations for processing all messages supported by the class. For example, the INTEGER class definition includes the addition and subtraction operations needed for manipulation of integers. In object-oriented terminology, those operations are needed for processing the addition and subtraction messages supported by the INTEGER class.

User defined built-in classes are not supported by Cyrano. Also, users may not request all members of a built-in class because some built-in classes, such as INTEGER, are open-ended, i.e., a query requesting all members of the INTEGER class will result in non-termination processing, and thus is not supported by Cyrano.

4.1.2 Gateway Classes

A member database structure translated into the Cyrano data model is of the gateway class. There is a one-to-one mapping between a gateway class and its corresponding structure in a member database. Gateway classes are therefore the Cyrano data model representations of the schemata of member databases.

In the context of deductive models, gateway classes are analogous to the extensional predicates. Finding answers to an extensional predicate requires the retrieval of data from the database via the predicate. Similarly, finding the members of a gateway class requires the retrieval of data from member databases via the gateway class. Gateway classes are also used by more complex classes, derived classes, to compute more complex data relationships, just as extensional predicates are used by intensional predicates to compute more complex relationships.

A gateway class defines the attributes and the methods of a class. A gateway object processes a message by issuing a request to the corresponding member database. Methods are not defined for a Cyrano gateway class which represents a structure in a non object-oriented database, because the concept of method is not supported by this kind of databases. The access to an attribute of such classes is, however, still via messages to the desired object. Cyrano implements these classes in such a way that any reference to any attribute of the instances of such classes is equivalent to sending a message to the corresponding object identified by an attribute name. Figure 6 shows an example of a gateway class. The gateway class *Student* corresponds to a relational table in which *Student* is the name of the table, and *name*, *ss_num*, *major*, *gpa*, and *sex* are the attributes of the table. The attributes of the class are all private. The names of the attributes are also

used as the names of the methods that access the attributes. For example, if *S* is defined to be a *Student*, then the reference *S.name* is a message to *S* in which the method *name* will retrieve the value of the object identified by *name.*

```
CLASS Student IS GATEWAY
    WITH
        STRING  name;
        STRING  ss_num;
        STRING  major;
        REAL    gpa;
        STRING  sex;
    END;
END CLASS.
```

Figure 6 Example of A Cyrano Gateway Class

4.1.3 Derived Classes

A derived class is derived from one or more gateway classes and/or one or more derived classes. A derived class contains a set of derivations. Each derivation defines a list of the base classes. These base classes could be either gateway classes or derived classes. Each derivation also contains a set of rules called a *guard.* The rules define the membership classification for the derived class.

A derived class also defines the attributes and methods of the class. A derived object processes a message by running the corresponding method defined for the derived class. Figure 7 shows some examples of derived classes.

```
CLASS TeachingAssistant IS DERIVED FROM
    Student S, Employee E: (S.name = (E.first_name + E.last_name))
    WITH
        STRING Name = E.last_name + E.first_name;
    END;
END CLASS.
```

(a)

```
CLASS GoodStudent IS DERIVED FROM
    Student S: (S.gpa > 3.0)
    WITH
        •••
    END;
END CLASS.
```

(b)

```
CLASS StudentParis IS DERIVED FROM
    Student S1, Student S2: ((S1.major = S2.major) AND
                             (S1.name <> S2.name))
    WITH
        Student First_Student: S1;
        Student Second_Student: S2;
    END;
END CLASS.
```

(c)

Figure 7 Examples of Cyrano Derived Classes

A derived class can combine two or more other classes as shown in Figure 7 (a). The class membership is specified via the rule *S.name = E.first_name + E.last_name*. The rule says that an instance of *Student* or an instance of *Employee* belongs to class *TeachingAssistant* if the name of the student instance is the same as the name of an employee instance. Here the name of an employee is assumed to be stored in two fields - first name and last name. Therefore, a concatenation operation is needed in order to compare an employee name with a student name.

A derived class can limit other classes as shown in Figure 7 (b). The *Good-Student* class limits its membership to those students with a GPA greater than 3.0.

A derived class can combine objects into composite objects as shown in Figure 7 (c). This class combines two student instances into a composite object, if the majors of the two students are the same.

Derived classes are used by Cyrano to resolve database heterogeneity. A derivation can normalize the values of the data from the objects of the base classes. A derived class can make data in different formats appear the same by using different derivations performing different normalizations. One example of data representation heterogeneity is shown in Figure 7 (a), where a student name is assumed to be stored in one field, the *name* attribute, in the format of first name plus last name. While an employee name is stored in two fields, the *first_name* attribute and the *last_name* attribute. The normalized *Name* field is assumed to be in the format of last name plus first name. The *Name* attribute gives a unified view of a name over two different representations of names such as those in the *Student* and *Employee* classes.

4.2 Queries and Query Language

Cyrano takes the query-by-class approach. Issuing a query is thus the same as defining a class. A query can either be written as a gateway class or a derived class. The query defined as the gateway class in Figure 6 will return all students in the member database corresponding to gateway class Student. The query written as a derived class in Figure 7 (b) will return all students with a GPA greater than 3.0.

The Cyrano query language is a formal language based on Domain Calculus [Ullm88]. Although the Cyrano research [Dzik96] does not define a calculus or an algebra as the query language, it does prove that the Cyrano query language fully supports the query languages of the relational data model, the Datalog (deductive) data model, and the core object-oriented data model defined in [Kim91].

The Cyrano query language is a language with predicates based on behavior. That is, the language views objects as instances of abstract data types (ADTs). Access to objects that are instances of an ADT is through public interface methods defined for that ADT.

The Cyrano query language allows the creation of new objects mainly due to its deductive nature. New objects are derived from the existing objects based on the deductive rules.

The Cyrano query language is an observing language. The language does not change the states of the member databases. All new objects created by the language are non-permanent, i.e., they will not be added to the member database as permanent data.

The Cyrano query language is non-restrictive. The language supports quantifiers and recursive queries.

4.3 Query Processing Methodology

Cyrano uses the most basic model of the deductive query processing methodology, the *Logic Query Expression - Logic Query Evaluation* methodology. This methodology is defined in Section 3.2. Figure 8 shows the Cyrano query processing methodology. The query processing takes a query, which is a class, and evaluates the query directly, without any optimization processing. The evaluation algorithm used by Cyrano is basically the Naive Evaluation algorithm which is the simplest deductive query evaluation algorithm. Naive Evaluation will be presented in detail in Section 6.1.2.1.

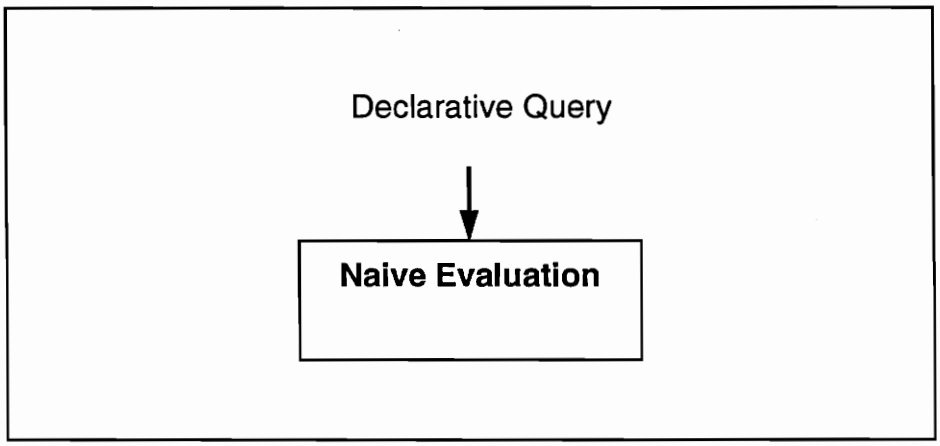

Figure 8 Cyrano Query Processing Methodology

Figure 9 breaks down the query processing into details. There are two main stages in Cyrano query processing, as indicated by the two dotted boxes. The first stage is the *compilation* stage, and the second stage is the *execution* stage. The following two subsections describe the two stages.

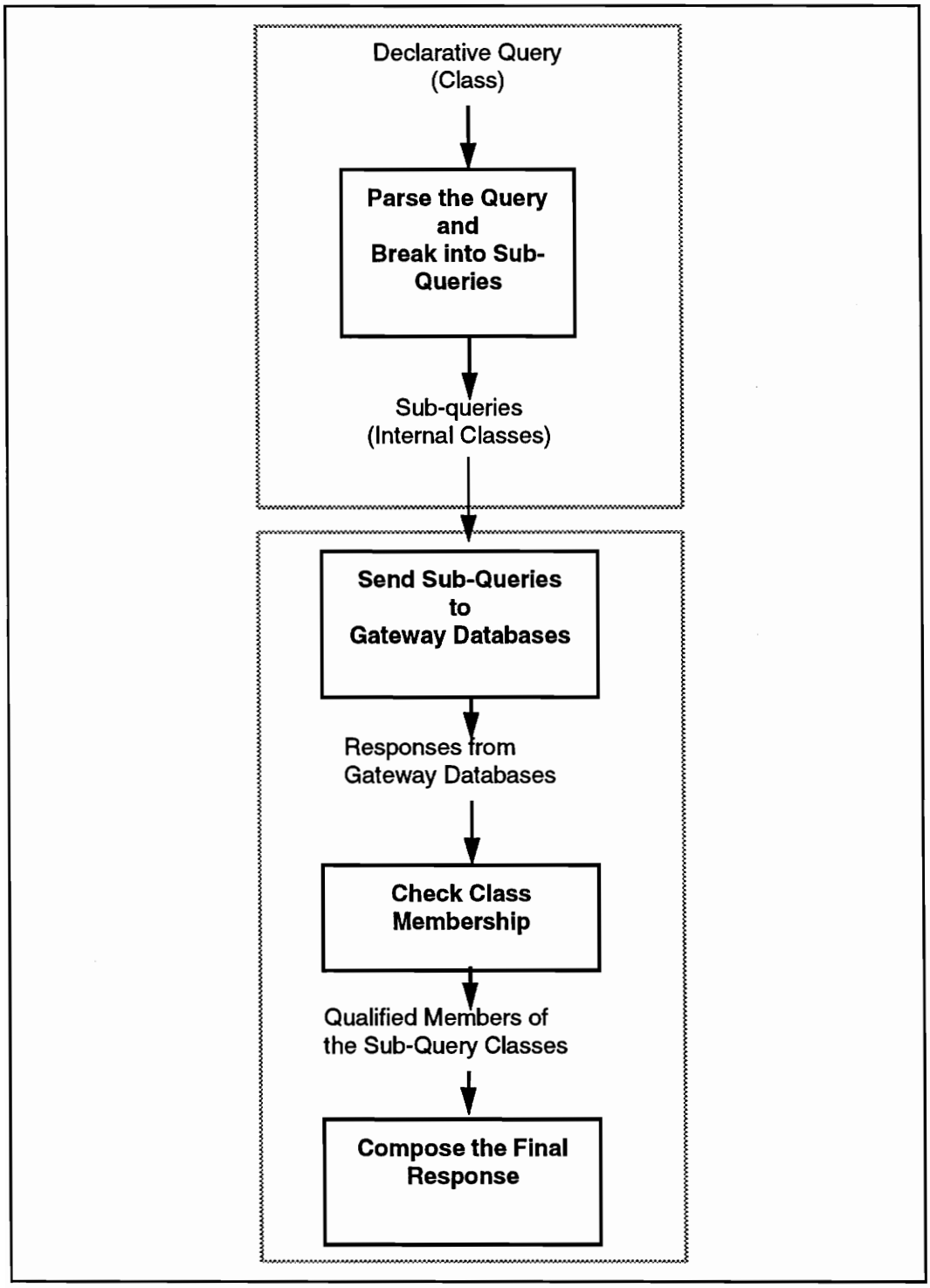

Figure 9 Cyrano Query Processing Steps

4.3.1 Compilation Stage

In the *compilation* stage, the query processor loads the query, which is expressed as a class, parses the query, and breaks it into sub-queries. The breakdown step is an iteration step which iterates through each base class until it reaches the gateway classes. The output of this stage is a set of sub-queries. The sub-queries are expressed as a tree of classes, in which each node represents a class. The root node is the original class, the query. Every other node in the tree represents a base class. A child node at level *n* represents a base class of its parent nodes at level *n-1*. Leaf nodes represent the gateway classes. All nodes above the leaf nodes represent derived classes. Figure 10 is an example of a query, *GoodTA*. The query finds all teaching assistants with a GPA of 3.5 or greater.

```
CLASS GoodTA IS DERIVED
    GoodStudent GS, TeachingAssistant TA:
    WITH
        •••
    END;
END CLASS.
where GoodStudent and TeachingAssistant are both derived classes.

CLASS GoodStudent IS DERIVED
    Student S: (S.GPA >= 3.5)
    WITH
        •••
    END;
END CLASS.
where Student is a gateway class.

CLASS TeachingAssistant IS DERIVED
    Student S, Employee E: (S.ss_num = E.ss_num)
    WITH
        •••
    END;
END CLASS.
where both Student and Employee are gateway classes.
```

Figure 10 A Sample Cyrano Query

Figure 11 shows the query tree of the sample query defined in Figure 10. The tree is the output of the *compilation* stage. The figure shows that the root, *GoodTA*, is the query. *GoodStudent* and *TeachingAssistant* are the base classes of *GoodTA*. *Student* is the base class of *GoodStudent*, and both *Student* and *Employee* are the base classes of the class *TeachingAssistant*.

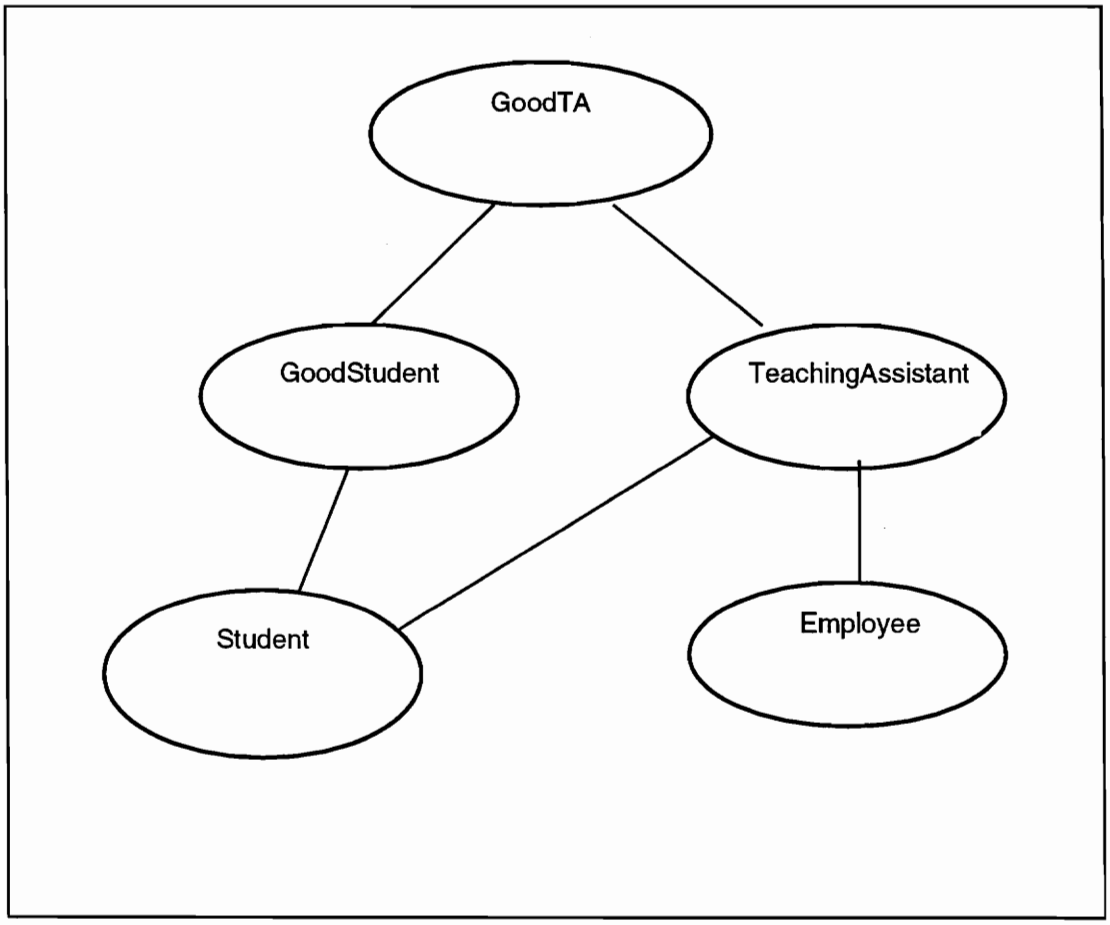

Figure 11 Tree for the Sample Cyrano Query

4.3.2 Execution Stage

In the *execution* stage, every sub-query is sent to a corresponding gateway database. The gateway database translates the sub-query into the languages of the corresponding member databases and sends the translated sub-query to the member databases. Responses from member databases are translated back to the global language and are forwarded back to the global database by the gateway databases. The global database checks the response objects against the rules to determine the qualified answer objects. The global database then composes the qualified objects into a global response and sends the response back to the user.

Cyrano uses the *bottom-up* evaluation method commonly seen in the deductive databases. Cyrano evaluates a query by repeatedly cycling through all base classes and the base classes of the base classes. The gateway classes are computed first, classes derived from the gateway classes are computed in the next cycle, and so on. Using the example shown in Figure 11, the order of the evaluation (computation) is as follows:

Student, Employee, GoodStudent, TeachingAssistant, and GoodTA.

Recursive queries are handled via repeated iterations. Recursive derivations are evaluated repeatedly against the new and existing objects. When no more new objects are derived from a cycle, the evaluation ends. This is basically a Naive Evaluation method. Figure 12 shows the Cyrano prototype query evaluation algorithm.

```
Done = FALSE;
DO UNTIL Done
    Done = TRUE;
    DO for all classes on the Result-List
        IF this is a gateway class
            IF this is the first time in the loop for this class
                Evaluate the class
                Add the result to the Result-List
                Done = FALSE
            ENDIF
        ELSE
            DO for all derivations of the class
                Find all combinations of the base objects of this class
                DO for all such combinations
                    IF a combination satisfies the guard of the derivation
                        Generate a derived object from the combination
                        Add it to the Result-List
                        Done = FALSE
                    ENDIF
                ENDDO
            ENDDO
        ENDIF
    ENDDO
ENDDO
```

Figure 12 Cyrano Prototype Query Evaluation Algorithm

One problem with this algorithm is that it does not guarantee termination for recursive queries. For a recursive query, deductive rules in the query will be recursively used, and the same objects may be repeatedly generated. The reason that the algorithm may not terminate is that as long as an object is generated, it will be added to the Result-List whether the object is already on the Result-List or not.

Consider a database with the following data and rules:

parent (Ken, Jim)

parent (Amy, Mary)

parent (Pat, John)

parent (Frank, Ken)

Rule 1: ancestor (X, Y):- parent (X, Y)

Rule 2: ancestor (X, Y):- ancestor (X, Z), parent (Z, Y)

where parent (X, Y) means *Y* is a parent of *X*. Rule 1 says that if *Y* is a parent of *X*, then *Y* is an ancestor of *X*. Rule 2 says if *Y* is a parent of *Z*, and *Z* is an ancestor of *X*, then *Y* is an ancestor of *X*. The rules are written in a logic language form.

Consider a query for finding all ancestors of Frank. The query is expressed, q (X):- ancestor (Frank, X). The following text lists the objects in each iteration of the loop and the contains of the Result-List after each iteration.

Iteration 1:

Object generated:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken)

Result-List:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken)

iteration 2:

Object generated:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Frank, Jim)

Result-List:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Frank, Jim)

Iteration 3:

Object generated:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Frank, Jim), (Frank, Jim)

Result-List:

(Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Frank, Jim), (Ken, Jim), (Amy, Mary), (Pat, John), (Frank, Ken), (Frank, Jim), (Frank, Jim)

• • •

As can be seen from the above example, the algorithm will not terminate. As long as objects can be generated, they will be added to the Result-List whether or not the objects are already on the Result-List.

The other problem with the algorithm is its inefficiency. The inefficiency is mainly caused by the nature of the Naive method, which will be presented in detail in Section 6.1.2.1.

5. Query Optimization for Object-Oriented Models

5.1 Challenges for Optimization of Object-Oriented Queries

An object-oriented model can support features such as abstract data types (ADT), methods, encapsulation, inheritance, complex structures, and object identity. While these features provide powerful modeling capabilities, they create new challenges for query processing in general and query optimization in particular. The following subsections discuss the query optimization problems introduced by these features. The discussion is based on [MZD94] and [MSTB94].

5.1.1 Abstract Data Types

Regarding ADT, [MZD94] considers three issues related to query optimization: namely *type specific* optimization, *subtype/subset* optimization, and *static type-checking*.

5.1.1.1 Type Specific Optimizations

Type specific optimization means that the optimization looks at relationships between objects of different types, and tries to simplify queries based on such relationships. *Type specific* optimizations are similar to those examined in the area of semantic query optimization. This kind of simplification (transformation) depends solely upon the information on the ADT. The following gives an example for the type specific optimization.

Query clause:

e.employer.name = v.manufactory.name

where e is an object of type Employee and v is an object of type Vehicle.

Axiom for ADT Company:

$\forall\ c_1, c_2$: Company c_1.name = c_2.name $\Rightarrow c_1 = c_2$

A *type specific* optimization could note that e.employer and v.manufactory both refer to the Company objects. The optimization applies the above axiom to the query and transforms the query to:

e.employer = v.manufactory

The simplified expression probably requires fewer object accesses since the *name* properties of the Company objects no longer need to be computed.

5.1.1.2 Subtype and Subset Optimization

Subtype and subset optimization uses the knowledge about the abstract data type construct. In particular, this type of optimization looks at the type - subtype relationships called *subtyping* relationships. Subtyping relationships give similar information about set inclusion relationships, i.e., all sets with member-type T are subsets of the instances of T. The following gives an example of using the subtyping relationships to simplify a query expression:

Answer:= Select (Vehicles, o v.manufactory.name = 'GM'
$\wedge$ v $\in$ GmEmpCars)

where o represents the return object(s).

The selection could be simplified to:

Answer := Select (GmEmpCars, o v.manufactory.name ='GM')

This expression eliminates the logical *AND* operation. The simplification is based on the knowledge that all sets with member-type T are subsets of the instances of T, i.e., GmEmpCars has type Vehicle, so it must be a subset of Vehicles, a subset of the instances of type Vehicle.

5.1.1.3 Static Type-Checking Issues

Sub-typing information could also affect the applicability of transformations in a statical type-checking system. For example, the query operation *Union* (Students, Employees) would normally result in a set having type Person, where Person is defined to be the closest common super-type of types Student and Employee. Thus, as far as type-checking is concerned, only properties of type Person are valid in the resulting set. This means that a query transformation which distributes operation *Union / Intersection / Difference* over other operations, such as a method *m*, is not applicable if static type-checking is enforced, e.g.,

Union (QueryOp (S_1, m), QueryOp (S_2, m)) $\neq$ QueryOp (Union (S_1, S_2), m)

since *m* may not be defined for the type of Union (S_1, S_2).

In order to allow such a transformation, some type inference mechanism is needed. The type inference mechanism of [Stra91] could be used to address this problem.

5.1.2 Complex Structures

The complex structure of objects means that languages which query the objects must have mechanisms for exploring their structures. Also, languages that support the creation of objects need mechanisms for building new structures. Supporting the exploration and creation of such structures can lead to the regular use of path expressions for navigating through a structure.

5.1.2.1 Path Expressions

One problem introduced by path expressions is that of the implied execution order on the path. This order, however, may not be the most efficient way to process the query. A path sometime can be more efficiently processed by using a *join*. Therefore, an optimizer needs to be able to make such transformations between path expressions and explicit *joins*. For example:

Query expression:

name (company (s)), where s is in Student class.

Corresponding path expression:

s.company.name

The path has the following execution order:

- apply the *company* method to *s*, and
- apply the *name* method to the result (from the preceding statement).

If there are very few companies it might be more efficient to first compute the *name* property for each company, store the result in a tuple, then *join* the tuple with *Students* by matching the *company* property of a student with the *company* attribute of the tuple.

5.1.2.2 Common Subexpressions

In general, optimization would utilize the common subexpressions in a query. However, in object-oriented systems, many expressions will be path expressions, which could complicate the optimization process because a common subexpression could be optimized differently over each instantiation of a common path. Thus, an optimizer must determine whether the optimization transformation is applicable to a single path leading to the subexpression, or to all paths. The authors of [MZD94] state that there is no research on deciding whether optimization should be applied to every occurrence of the common subexpression.

5.1.3 Methods

In general, the cost information about objects is necessary in deciding about the applicability of an algebraic transformation. The determination of the cost is complicated by the presence of methods and encapsulation. Methods could be implemented by arbitrary computations, which make transformation of expression difficult; and encapsulation prevents an optimizer from obtaining the implementation details of the methods.

5.1.3.1 Transformation

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 can optimize the method code as a nested query. However, query languages in OODBMSs can access arbitrary methods defined for abstract data types, and these methods may be written in languages not recognized by a query optimizer. Research done in this area will be discussed in Section 5.2.2.

5.1.3.2 Cost Estimation

One approach is to let the optimizer break the encapsulation and look into the implementation of the methods. This approach requires a method written in the query language that is understood by the optimizer. The method code could then be merged with the query and managed by the optimizer. This approach, however, limits the expressiveness of methods to that of the query language.

Another approach is to let the optimizer query a method to obtain the cost of the method. This requires that every method define an interface that can provide the required information. An alternative way to determine the cost of a method would be to store precomputed results. The costs of the method application are then transferred to compilation time.

5.1.4 Encapsulation

Encapsulation makes it difficult for an optimizer to get the "big picture." Current (relational) query optimization assumes that a complete description of the query is available, in terms of structural operators. In an object-oriented database, a query may be as simple as a single message to a single object. In that case, the range of transformations the optimizer can apply is limited because the expressions to be optimized are very small.

Another problem with encapsulation is due to the multiple implementations of a type. Relational set processing depends a great deal on the homogeneity of structures, as during the allocation of temporary storage space of records in intermediate results or in the computing of sets for fields in records. However, in object-oriented databases, a collection of elements of type T could have heterogeneous structures, because of the possibility of multiple representations in the various implementations of T. Thus an optimizer would have to be able to determine the applicability of transformations wherever the heterogeneity problem is involved.

5.1.5 Object Identity

Object identity (OID) can affect the optimization of queries. When objects have identities, there is a question as to what constitutes the equality of two objects. The issue is further complicated by the new objects. The creation of new objects can lead to new definitions of equivalence of queries that affect transformations available to an optimizer. Thus, an optimizer in object-oriented systems must be able to deal with the creation of new objects and with alternative definitions for equivalence. The issues are addressed in the following subsections.

5.1.5.1 Object Equality

Equality of objects in a query can refer to any definition of equality of type *Object* (e.g., identical), or to equality operations defined for a particular abstract data type. In a query language, a variety of equality operators may be permitted in predicates. Some examples are the *shallow* and *deep* equality operators. Two objects are said to be *shallow-equal* if their values are identical. Two objects are said to be *deep-equal if*: (1) they are atomic objects and their values are equal; (2) they are set objects and their elements are pair-wise deep-equal; or (3) they are tuple objects and the values they take on the same attributes are deep-equal.

When a language allows the creation of new objects, there are two possibilities regarding object identity: (1) the language creates new objects without new identities; or (2) the language creates new objects with identities. In the first case, the equality operations between two new objects and the equality operations between a new object (without OID) and an existing object (with OID) could be different. In the second case, the creation of new objects may result in the creation of new equality operations.

Therefore, in the presence of an OID, an equality operation is actually a method and should be treated as such by the optimizer. In addition, in a language which supports the creation of new objects, the optimizer may need a mechanism for deciding whether to create identities for objects in the intermediate stages of query processing.

5.1.5.2 Object Equivalence

The creation of objects with identity complicates the meaning of the equivalence of queries. The answer to a query can be a new collection of objects with a unique identity. As a result, even two responses to exactly the same query may not be identical. A weak 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 creation of new objects by a query language means that the structure of the result as well as the data retrieved by a query must be considered when defining equivalence. On the other hand, the ability to define an alternative equivalence might allow the optimizer to choose a more efficient method for solving a query.

5.1.5.3 Common Subexpression Equivalence

The creation of objects by a query complicates the determination of whether two query expressions are the same. If a query expression represents a constant or a variable, there is straightforward way to determine common subexpressions. For example, in the query

join (People, People, etc.)

It is clear that both references to People refer to the same database set.

The equivalence issue is further complicated by queries involving methods which create new objects. For example, in the query

join (Q, Q, etc.)

where Q has methods which create new objects, the two executions of the Q sub-query could create different objects, and thus may not be considered to be common subexpressions.

In general, an optimizer should be able to ignore the different objects generated by multiple applications of a sub-query expression. On the other hand, it is possible that an optimizer may make use of such occurrences if their presence can lead to more efficient optimization.

5.2 Optimization Techniques for Object-Oriented Models

Much of the work in object-oriented query optimization techniques is focused on the physical level, i.e., to find efficient ways to access information referenced by a path expression. The main techniques used in object-oriented query optimization are *indexing* [BK89, KM90, Bert94], *clustering* [BD90, PZ91, CD92], and *path expression manipulation* [JWKL90, LVZ92].

Other works are focused more on the logical level in which the main technique is query expression rewriting / transformation. Since this research is focused on the logical level query optimization, logical level optimization techniques will be discussed, and physical level optimization techniques will be mentioned only when appropriate.

5.2.1 Query Rewrite / Transformation

A number of proposals have been made for object-oriented optimization based on the algebraic rewrite of query expressions. Representative works in this area are described below.

Straube and Ozsu [SO90] define both syntactic and semantic rewrite rules for the object algebra. The rules 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. [SO90] will be described in detail in Section 5.3.3.

Finace and Gardan [FG91] present a rule language for specifying query rewriting. The language allows writing rules that describe both syntactic and semantic transformations. The authors also provide a meta-rule language that allows an implementation to define blocks of rules and sequences of rules. Meta-rules help define which rules should be applied at which point, and therefore simplify the rule search strategy [MZD94].

Beeri and Kornatzky provide some 40 rules which can be used for the purpose of rewriting [BK93]. Some of the rules are also extended to deal with recursively defined data structures. [BK93] mainly focuses the optimization effort on improving the processing of iterations over data collection, which is a predominant factor in bulk data processing. Since [BK93] algebra is an FP-like language, all rules are expressed on the functional level without referring to objects.

5.2.2 Method Optimization

One area of research focuses on the optimization of method code. The following give some representative works in this area.

Graefe and Ward [GW89] present a system which statically generates query evaluation plans with alternatives. Information available on methods is used at execution time to choose among the alternatives and generate a final evaluation plan.

Bertino proposes to precompute the results of methods, and store these results using an index [Bert91]. In [Bert91] a method over an object O can be written in an arbitrary language, but can not have input parameters other than O. Also, the method can not have side effects and must only use the primitive properties of O, i.e., the properties whose values are stored as part of O. These requirements de-

termine whether a precomputed result of a method is valid. If the result is valid, the result can be retrieved using the index. If the result is invalid, the method will need to be computed at query execution time.

Daniels et al. [DGKM91] present the REVELATION architecture. This architecture has an optimizer in which methods *reveal* information about their execution. The revealed information is then used to expand the nodes of a query tree. The query tree, when fully expanded, could be used as the input to a rule-based optimizer such as the one generated by Volcano [GCDM94].

5.3 Some Object-Oriented Optimization Systems

This section examines some object-oriented query optimization systems, in particular, the unification of the rewrite-based and type-based optimization techniques of Cluet and Delobel [CD94], the Epoq architecture of Mitchell, Zdonik, and Dayal [MZD94], and the query processing system of Straube and Ozsu [SO90, Stra91].

The Straube and Ozsu query processing system (SOSYS) will be presented in detail because this system is rich in object-oriented data model concepts, object calculus, object algebra, and includes a comprehensive query processing methodology that ranges from logical query transformation to physical plan generation.

5.3.1 A Unified Rewrite-Based and Type-Based System

Cluet and Delobel propose a formalism that provides for the integration of type-based query optimization with rewrite-based optimization techniques. Their approach is to introduce types in algebraic expressions and to reduce complex ex-

pressions representing *selection*, *projection*, and *join* operations. The result is a model that integrates rewrite techniques for paths, algebraic query rewrite, and common subexpression factorization.

The combination of these different techniques causes the generation of a large number of possible expressions from a simple query. Index and clustering information are normally used to reduce the search space for equivalent expressions.

5.3.2 Epoq System

The Epoq system is motivated by the need to integrate a variety of strategies for solving different problems in query optimization. An Epoq optimizer is a collection of concurrently available region modules, each of which embodies one strategy of the optimization of query expressions. Different regions often accomplish different query transformation tasks, but regions may also represent different strategies for accomplishing the same task in different ways. The Epoq architecture integrates the regions through a common interface for region modules, and a global control that combines the actions of subordinate regions to process a given query.

The region modules are organized hierarchically, with a parent region controlling its subordinate regions as though they were a collection of transformations. The regions define, through their interface, the characteristics of queries they can process, goals for the transformation of queries, and the characteristics of result queries. A higher-level control uses this information to plan a sequence of region executions to process a given query expression. The control is a goal-directed

planner that intermingles the planning of the optimizer's processing with the execution of region modules, and uses execution results to direct further planning of the optimizer's processing.

The Epoq system integrates diverse strategies for query optimization and allows the addition of new strategies. Epoq expands the query optimizers to include extending the collection of optimization strategies that can be applied in the transformation of a query expression. Thus an Epoq optimizer can incorporate new strategies that are developed to solve problems in object-oriented query optimization.

5.3.3 Straube and Ozsu System

The Straube and Ozsu system (SOSYS) [SO90, Stra91] defines an object-oriented data model, an object calculus, and an object algebra. The calculus and algebra are used as query languages. The system provides a comprehensive query processing methodology by extending the relational query processing methodologies.

5.3.3.1 Data Model

SOSYS defines an object-oriented data model. All definitions in the model are based on the existence of the following sets:

- A finite set *SD* of basic domains D_1,..., D_n where $SD = \bigcup_{i=1}^{n} D_i$, and D_i is one of the basic domains such as integer, string, etc.
- A countably infinite set *A* of symbols called attributes.
- A countably infinite set *ID* of identifiers.
- A finite set *CN* of class names.
- A finite set *MN* of method names.

SOSYS supports the basic object-oriented features of object identity, class, class hierarchy and inheritance, aggregation and generalization, and encapsulation.

5.3.3.1.1 Values

SOSYS defines three types of values:

1. Every element in *SD* is an atomic value.

2. Every finite subset of *ID* is a set value.

3. Every element in IP(A) x IP(SD) x IP(ID) is a structural value, where *IP (x)* denotes the power set of *x*.

The symbol *V* denotes the set of all values.

5.3.3.1.2 Objects

SOSYS defines an object *o* as a triple:

o = (id, cn, val), where id $\in$ ID, cn $\in$ CN, and val $\in$ V.

O = ID x CN x V is the set of all objects. Notations *o.id*, *o.cn*, and *o.val* denote the identifier, the class, and the value of an object *o*, respectively.

SOSYS defines the notion of consistent sets of objects, which are basically the concepts of *unique identifier* and *referential integrity*. The definition is as follows:

A set of objects Θ is consistent if and only if:

- $\forall$ o, p $\in$ Θ, o.id $\neq$ p.id - No two objects in Θ have the same identifier.
- $\forall$ o $\in$ Θ, ref (o) $\subseteq \cup_{p \in \Theta}$ p.id - Each identifier in ref (o) is an object in Θ,

where *ref (o)* is defined as the association to an object *o* with the set of all identifiers referenced within the value part of the object. It is a recursive function which includes all references of objects nested within *o*.

5.3.3.1.3 Methods

SOSYS defines a method m as a triple:

m = (mn, f: S $\rightarrow$ T, b) where:

- *mn* $\in$ MN is the name of the method.
- *f*: S $\rightarrow$ T is a function mapping a product of source domains to a target domain in the form of:

  f: S_1 x S_2 x... x S_n $\rightarrow$ T, where S_1,... S_n and T are sets of objects.
- *b* is the behavior (semantics) of function f.

Function *f* is n-ary, which allows parameterized methods. The general form of *f* is

f (s_1,..., s_n) = t, where $s_1 \in S_1$,..., $s_n \in S_n$ and t $\in$ T.

In object-oriented terminology, the above states that function *f* is applied to object s_1 using objects s_2,..., s_n as parameters, resulting in object *t*. The method name, the number and type of the arguments, and the type of the result, comprise the signature of the method.

5.3.3.1.4 Classes

SOSYS views classes as abstract data types (ADTs) whose instances are objects. In SOSYS, a class is a type definition as well as a synonym for all objects which are instances of the type. SOSYS defines a class *c* as a triple:

c = (n, p, MT), where

- $n \in$ CN is the class name.
- p is a sequence of parent classes of the form $<c_1,..., c_k>$.
- MT is a set of methods.

A class defines the interface and semantics of its instances by making public a set of methods and their signatures.

SOSYS supports class hierarchy and class inheritance. The inheritance is behavioral inheritance, i.e., a subclass of a class behaves like its parent class. This is to ensure the conformity or substitutability. Substitutability ensures that an object of a class *C* can be used in any context specifying a superclass of *C*.

SOSYS supports multiple inheritance, but views the multiple inheritance conflict resolution protocol as an implementation issue. The authors suggest the use of one of two alternatives: (1) requiring the user to specify the multiple inheritance semantics, or (2) resolving the conflict according to a predetermined ordering of the class lattice.

5.3.3.1.5 Databases

SOSYS defines a database *db* as follows:

db = (C, Θ)

where *C* is a set of classes denoting a class hierarchy, and Θ is a consistent set of objects meeting the following constraints:

- $\forall c \in$ C, the name of c is unique.

- $\forall$ o $\in \Theta$, o.cn is the name of a class in C.
- The root of the class $c_{root} \in C$.
- $\forall$ c $\in$ C, c is a subclass of c_{root}.

The first two constraints ensure that each object in the database belongs to only one class, which is itself in the database. The third and fourth constraints ensure that the recursive definition of inheritance terminates for every class in the database.

SOSYS defines a database operation as a function:

op: db x $<o_1,..., o_n>$ x mn $\rightarrow$ r, where

- *db* is a database.
- $<o_1,..., o_n>$ are objects in the *db*.
- *mn* is the name of a method defined for the class of o_1.
- *r* is the resulting object.

5.3.3.2 Queries and Query Languages

SOSYS focuses on formal query languages, such as object calculus and object algebra. The languages support strict encapsulation, i.e., operators which manipulate or depend on the object representation are not allowed. The languages take the approach of predicates based upon behavior. The languages are object-preserving languages.

5.3.3.2.1 Query Primitives

SOSYS takes an object-oriented approach and views queries as classes. However, queries in the SOSYS are not defined in the class format but rather as a combination of atoms and operators. These atoms and operators are the building blocks of the object calculus.

Atoms

Atoms are the building blocks of calculus expressions and predicates for qualifying algebra operators. They represent the primitive query operations of the data model and return a boolean result. Three types of atoms are defined in the SOSYS.

1. o_i Op o_j: where o_i and o_j are either object variables or denote an operation of the form $<o_1,...,o_n>$.mlist where $o_1...o_n$ are object variables, *mlist* is a list of methods, and *Op* is one of the operators defined under *Operators*.

2. a Op o_j: where o_j are either object variables or denote an operation of the form $<o_1,...,o_n>$.mlist where $o_1...o_n$ are object variables, *mlist* is a list of methods, *a* is the textual representation of an atomic value or a set of atomic values, and *Op* is one of the operators defined under *Operators*.

3. Range atom: C(o) or C*(o) where *C* is the name of a class and *o* is an object variable ranging over the instances of class *C*. There are two cases:

   - C(o): refers to the objects in the *extent* of *C*.
   - C*(o): refers to the objects in the *deep extent* of *C*.

where *extent of C* specifies the instances of *C*, and *deep extent of C* refers to all instances in the *extent* of all classes rooted at *C*.

Operators

There are four comparison operators defined:

1. Object identity equality: ==

2. Set value inclusion: $\in$

3. Set value equality: $=_{\{\}}$

4. Atomic value equality: =

These four operators can be used directly in queries. In addition, logical operators such as $\neg$, $\wedge$, $\vee$, and quantifiers such as $\exists$ and $\forall$ can also be used directly in queries. Any other operators must be implemented as methods. The following is a query for finding every person *p* with an age greater than 65.

"65" = q $\wedge$ "true" = <p, q>.age.greater.

where "=" is the atomic value equality operator and *greater* is implemented as a method.

5.3.3.2.2 Object Calculus

The SOSYS object calculus is a declarative query language. The format of the object calculus definition is similar to the tuple relational calculus definition provided in [Ullm88]. A query in the object calculus is in the form of

{o | ψ (o)},

where o is an object variable denoting some object in the database, and ψ is a formula built from atoms. The result of a query is a set of objects o which satisfy the predicate formed by $\psi(o)$.

Before defining formulas, the notion of *free* and *bound* variables needs to be introduced. A variable is said to be *bound* in a formula if it has been quantified via quantifiers such as $\exists$ or $\forall$; otherwise, the variable is *free* in the formula. The definitions of the formulas are as follows:

1. Every atom is a formula. All object variables in the atom are free in the formula.

2. If ψ_1 and ψ_2 are formulas, then $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, and $\neg\psi_1$ are formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, and $\neg\psi_1$ as they are free or bound in ψ_1 or ψ_2.

3. If ψ is a formula, then $\exists o\ (\psi)$ is a formula. Free occurrences of *o* in ψ are bound to $\exists o$ in $\exists o\ (\psi)$.

4. If ψ is a formula, then $\forall o\ (\psi)$ is a formula. Free occurrences of *o* in ψ are bound to $\forall o$ in $\forall o\ (\psi)$.

5. Formulas may be enclosed in parentheses. In the absence of parentheses, the order of precedence is $\in$, =, $=_{\{\}}$, ==, $\exists$, $\forall$, $\neg$, $\wedge$, $\vee$, where $\in$ is the highest in the order.

The formal definition of a query in the object calculus is then:

$\{o \mid \psi\ (o)\}$, where *o* is the only free variable in ψ.

SOSYS further provides checks for guaranteeing the safety of a query expressed in the object calculus. The safety of a query is defined as follows:

> An object calculus expression is considered to be *safe* if the expression can be evaluated in finite time and produces finite output.

5.3.3.2.3 Object Algebra

The SOSYS object algebra is a procedural query language. The object algebra implements a subset of the object calculus, namely the restricted class of object calculus expressions. The restricted class of object calculus expressions is defined as:

> Any safe object calculus expression of the form $\{o \mid \exists p \exists q \exists r \ldots \psi(o, p, q, r, \ldots)\}$,

where ψ does not contain any occurrences of either $\exists$ or $\forall$.

This class of queries is similar in power to the *select-project-join* class of queries in the relational model. The main exclusions from the class are the universal quantification and recursion such as those often found in rule based models.

The algebra defines five operators:

Union ($P \cup Q$):

The *union* is the set of objects which are either in *P*, in *Q*, or in both *P* and *Q*. An equivalent calculus expression for *union* is $\{o \mid P(o) \vee Q(o)\}$.

Difference (P - Q):

The *difference* operation produces a set of objects which are in *P* but not in *Q*. An equivalent calculus expression of *difference* is {o | P (o) $\vee \neg$Q (0)}. The *intersection* operator, P $\cap$ Q can be derived by P - (P - Q).

Select (P σ_F *<Q1...Qk>)*:

Select returns the objects denoted by *p* for each vector <p, q_1,..., q_k> $\in$ Px Q_1 x...x Q_k which satisfies the predicate *F*. An equivalent calculus expression of *select* is {p | P(p) $\wedge$ $Q_1(q_1)$ $\wedge$... $\wedge$ Q_k (q_k) $\wedge$ F (p, q_1,..., q_k)}.

Generate (Q_1 γ^t_F <Q_2... Q_k>):

This operation returns the objects denoted by *t* in *F* for each vector <q_1,..., q_k> $\in$ Q_1 x... x Q_k which satisfies the predicate *F*. *F* is a predicate with the condition that it must contain one or more generating atoms for the target variable *t*, and *t* does not range over any of the argument sets.

Two common uses of the *generate* operator are to collect results of method applications and to iterate over the content of set valued objects. An equivalent calculus expression of *generate* is {t | Q_1 (q_1) $\wedge$... $\wedge$ Q_k (Q_k) $\wedge$ F (t, q_1,..., q_k))}.

Map ($Q_1 \rightarrow_{mlist}$ <Q_2... Q_k>):

Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in <Q_2... Q_k> as parameters to methods in *mlist*, where *mlist* is a list of method names in the form of m_1,..., m_m. The operation returns the set of objects resulting form each sequence method application. *Map* is a

special case of the *generate* operator. This form of the *generate* operation supports several useful rules for optimization. An equivalent calculus expression of *map* is $\{t \mid Q_1\ (q_1) \wedge ... \wedge Q_k\ (q_k) \wedge t == <q_1,..., q_k>.\text{mlist}\}$.

5.3.3.3 Query Processing Methodology

SOSYS applies relational techniques to produce a methodology for query processing. The methodology is very similar to that proposed by Jarke [JK84], with the addition of type checking. Figure 13 shows the SOSYS query processing methodology.

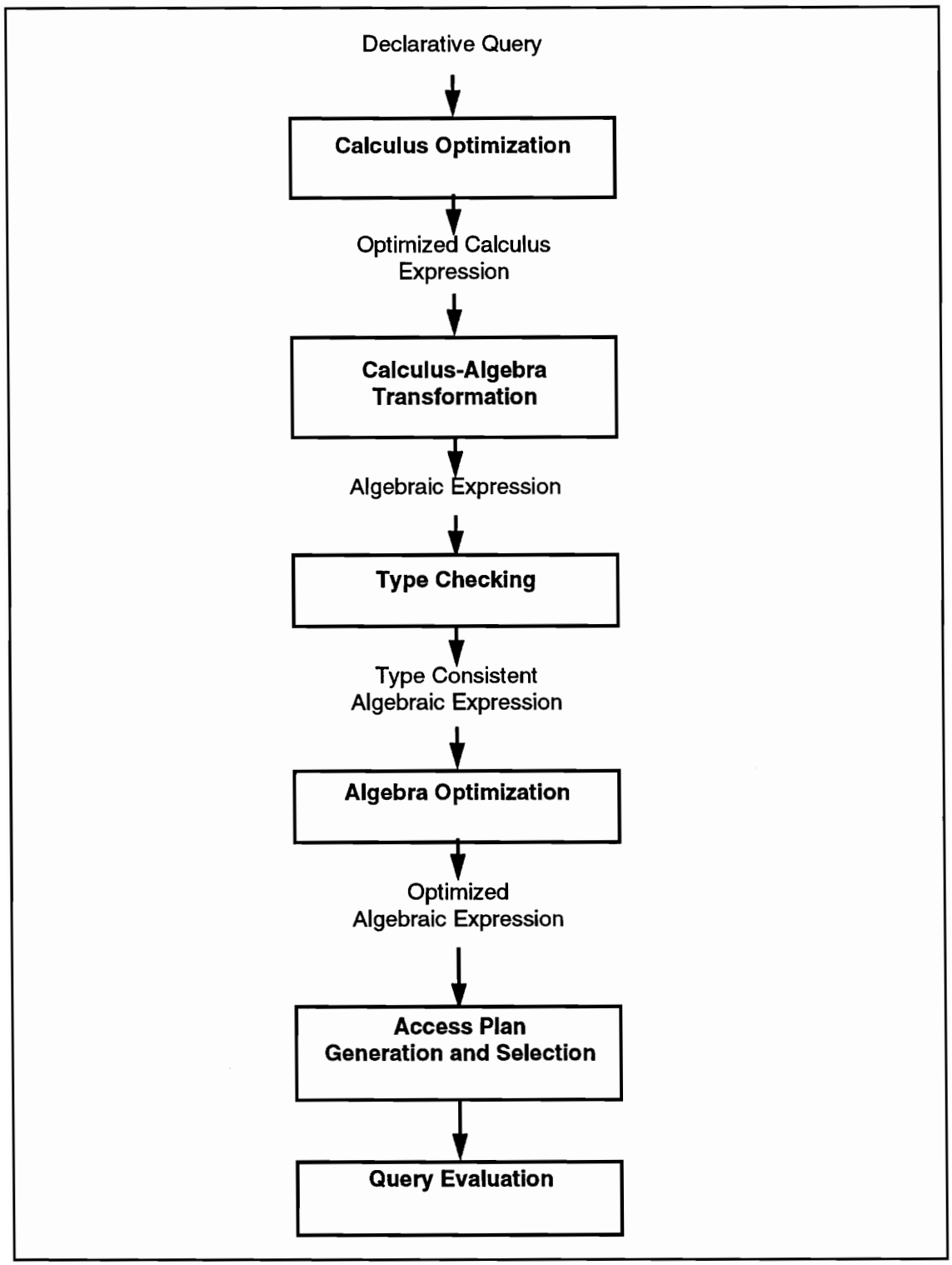

Figure 13 SOSYS Query Processing Methodology

The system takes a declarative object-oriented query expressed in object calculus as input and performs calculus optimization. The optimized calculus expression is then translated to the equivalent algebraic expression. Type checking is applied to the algebraic expression to guarantee type consistency. Algebra optimization is then applied to the type-consistent algebraic expression. Finally access plans are generated, and the optimal plan is used to produce an answer for the query.

5.3.3.3.1 Calculus Optimization

There are two main purposes of calculus optimization in the SOSYS. The first is to reduce the calculus expression to a normalized form in order to perform the calculus-to-algebra transformation, and the second is to perform a safety check in order to guarantee the safeties of the calculus expression before the calculus-to-algebra transformation. These processes, together with the calculus-to-algebra transformation, are included in one algorithm. The algorithm is called Calculus-to-Algebra Translation, and is presented in Figure 14.

```
Input:   Object calculus expression: Cexp
Output: Equivalent object algebraic expression.

begin
    convert calculus expression to prenex disjunctive normal form
    for each disjunct do
        for each constant defining atom of the form v = const do
            delete the atom and replace all other occurrences of v by const
        endfor
        rewrite all atoms using the internal notation
        for each atom a (v | r1,..., rn) for which there exists a range atom for v of the form
         a (v | _) do
            move v to the right hand side of the atom
        endfor
        for each atom of the form a (g | g, r1,..., rn) do
            rewrite the atom as a(_ | g, r1,..., rn)
        endfor
        call Place (t), where t is the target variable of Cexp resulting in Cexp'

        from innermost to outermost parentheses nesting of Cexp' do
            map the expression to equivalent algebraic expression
        endfrom
    endfor
    combine the algebra operator trees for each disjunct using Union
end.


Place (t)
Input:   target variable t
         list of atoms
Output: safe/unsafe query indicator
         nested expression of atoms which can be directly mapped to algebraic expression
begin
    The details of the function are not given here for the purpose of simplicity
end.
```

Figure 14 SOSYS - Calculus-to-Algebra Translation Algorithm

The algorithm inserts matched pairs of parentheses into the disjuncts of the Prenex Disjunctive Normal Form (PDNF) of a query. The algorithm terminates successfully when the innermost nested expressions correspond to the range atoms of the query. If this is not the case, then the query is *unsafe*. After rewriting, nested subexpressions are mapped directly to their object algebra counterparts. An example using the algorithm is given in Appendix B.

5.3.3.3.2 Type Checking

SOSYS introduces this new phase into the traditional query processing methodology. Traditional database query languages require minimal type checking because of the limited number of primitive domains (e.g., integer, string, boolean) supported by the traditional data models. Object-oriented query languages introduce complexity into this process since query results may be non-homogeneous sets of objects.

SOSYS developed a rather comprehensive object-oriented type checking procedure. The system provides type conformance definitions. This type of definition identifies those objects which may have different types but nevertheless support a common set of operations. The system provides a set of inference rules which utilize the definitions in order to determine the conformity of the objects.

5.3.3.3.3 Algebra Optimization

Algebra optimization in SOSYS is mainly accomplished in query rewriting. SOSYS provides equivalence-preserving rewriting rules for object algebraic expressions. Both structural and semantics rules are provided. Structural rules create equivalent expressions based on pattern matching and textual substitution. Se-

mantic rules are similar, but they are additionally dependent upon the semantics of the database schema as defined by the class definitions and inheritance lattice. The full set of rules is presented in Appendix C.

Straube gives two heuristics to be used to drive the rewriting process. The first is to push operations which reduce intermediate result sizes as far down the tree as possible. The result is that operand sets are reduced as early as possible, thereby minimizing the input to operations higher up in the query tree.

The other is to eliminate redundant or useless cross product generation. The semantics of *select*, *generate*, and *map* operations require that predicate or method applications in the *map* be evaluated for each element in the cross product of the argument sets. For example, the predicate *F* in P σ_F <Qset, Rset> is evaluated once for each vector <p, $q_1 ... q_k$, $r_1 ... r_l$> $\in$ P x Q_1 x... x Q_k x R_1 x... x R_l.

If *F* is broken down into two sub-formulas F_1 (p, $q_1 ... q_k$) and F_2 (p, $r_1 ... r_l$), then the *r* elements of the vector are not used in evaluating F_1, and the *q* elements are not used in evaluating F_2. Assuming the cost of generating the cross product grows non-linearly with respect to the number of sets, where the number of sets is *k+l+1*, it will cost less to generate smaller cross products P x Q_1 x... x Q_k and P x R_1 x... x R_l than to generate P x Q_1 x... x Q_k x R_1 x... x R_l. Also, the breaking down may make parallel processing of the products possible when such facility is provided.

The rewriting process may produce more than one algebraic expression from the original algebraic expression. Those new expressions will be processed further by the access plan generator to generate the least cost access path. The generation of an access plan is a physical level processing activity and will not be discussed here.

6. Query Optimization for Deductive Object-Oriented Models

Query languages in deductive object-oriented models are usually logic languages (e.g., Datalog) with object-oriented extensions. Query processing in deductive object-oriented models usually involves solving a *goal*, based on a set of rules (Intensional database - IDB) and a set of facts (extensional databases - EDB). As a consequence, query evaluation and optimization techniques developed for the deductive data models are in general applicable to deductive object-oriented data models. Since deductive object-oriented data models are also object-oriented data models, the challenges in optimizing object-oriented queries, as presented in previous sections, are also applicable to deductive object-oriented models.

Since Datalog is a widely used query language in deductive databases, and since Datalog-like languages are used for deductive object-oriented databases (e.g., O-Telos for ConceptBase [JGJS95]), the following subsections will use Datalog to illustrate query optimization techniques.

A general Datalog statement has the following format:

LHS:- RHS

where LHS stands for the left-hand-side of the rule, and RHS stands for the right-hand-side of the rule. LHS is also referred to as the *head* of the rule, and RHS is also referred to as the *body* of the rule. In general, an IDB rule has both the LHS and the RHS, e.g., *ancestor (x, y):- parent (x, y).* An EDB rule has one side only, e.g., *parent (John, James).*

A brief introduction to the Datalog language is provided in Appendix D. A comprehensive introduction to the language can be found in [CGT90].

6.1 Optimization Techniques for Deductive Object-Oriented Models

There are two different approaches in solving a Datalog query: *top-down* or *bottom-up* evaluation. The evaluation of a Datalog query involves building a proof tree. The concept is to view the answering of a query as a proof in logical theory. The tree can be constructed either from the goal (root) or from the leaves. Top-down evaluation starts from the goal, and tries to verify the premises which are needed for the goal to hold. Bottom-up evaluation starts from the existing facts and infers new facts, thus proceeding toward the goal. These evaluation mechanisms are also called *search strategies*, in that the mechanisms are used to search for the solutions.

The following subsections briefly introduce the evaluation / optimization algorithms used in deductive databases.

6.1.1 Top-Down Evaluation

In *top-down* evaluation, rules are seen as problem generators. Each goal is treated as a problem that needs to be solved. The initial goal is unified with the left-hand-side of some rule, and other problems are generated which correspond to the right-hand-side literals of that rule. This process is continued until the proof tree is completed. After the proof tree is completed, if the goal contains some bound argument, then only facts that are related to the goal constants are involved in the computation. Therefore, top-down evaluation performs a relevant optimization, be-

cause the computation automatically disregards many of the facts which are not useful for producing the result. The disadvantage of *top-down* evaluation is that the method produces the answer one object at a time, rather than one set of objects at a time. Answering with one set at a time is usually more desirable for query processing, as in this example of a very common query: find all students who work for IBM full time and attend graduate school part time.

The *top-down* approach can be further divided into two search methods: *depth-first* and *breadth-first*.

In the *depth-first* search method, the evaluation method generates subproblems corresponding to the right-hand-side of a rule according to a certain order of the right-hand-side literals, e.g., some algorithms use an order which always selects the right most literal, while other algorithms use an order which always selects the left most literal. Regardless of the order of selection, the evaluation will produce trees which grow in depth. The problem of the *depth-first* approach is that the chosen order of processing literals in rule bodies strongly affects the performance and even the termination of the evaluation. For example, Prolog uses the *depth-first* approach with the left-to-right order of literal selection. The performance of Prolog is therefore determined by how close the goal is to the left-most literals. In addition, Prolog does not guarantee the termination of the evaluation. The termination problem is left in the hands of the programmers [CGT90].

With the *breadth-first* search, the generation of all subproblems from the right-hand-side of a rule is done at the same time, thus producing a balanced growth of the search tree. Datalog goals seem to be more naturally executed

through b*readth-first* techniques, because the results of the computation in this method are neither affected by the order of predicates in the right-hand-side of rules nor by the order of rules within the program.

The *top-down* approach usually uses resolution refutation techniques to solve a goal. Nillson gives an excellent description of the resolution refutation technique [Nils80]. The specific resolution technique used in answering Datalog queries is called SLD-resolution. SLD stands for "Linear Resolution with Selection Function for Definite Clauses." SLD-resolution does not stand for a particular algorithm but for an entire class of algorithms. [CGT90] states that among different resolution methods, this class is the most appropriate for answering Datalog queries.

6.1.1.1 SLD-Resolution

The basic resolution method is to use counter-examples to disprove the negation of a theory, thereby proving the theory. In the context of the resolution method, a goal (or query) is a negative clause which needs to be disapproved by the method. For example, there are a set of Datalog rules R, an EDB, and a query Q (X):- p (a, X). The goal represents the assertion $\forall$X: $\neg$ p (a, X). The resolution method tries to disprove this assertion by generating counter-examples by finding constants A via R and EDB such that p (a, A) is true. The existence of such A contradicts the assertion $\forall$X: $\neg$ p (a, X), and each object $\in$ A is therefore a counter-example of the goal. The set of objects in A is thus the solution to the query.

In the resolution method, each different way of disproving the goal gives a one-object answer to the query. Each disproving process is called a refutation of the goal G from R ∪ EDB, and each such process generates a linear refutation tree. Figure 15 shows a general linear refutation tree.

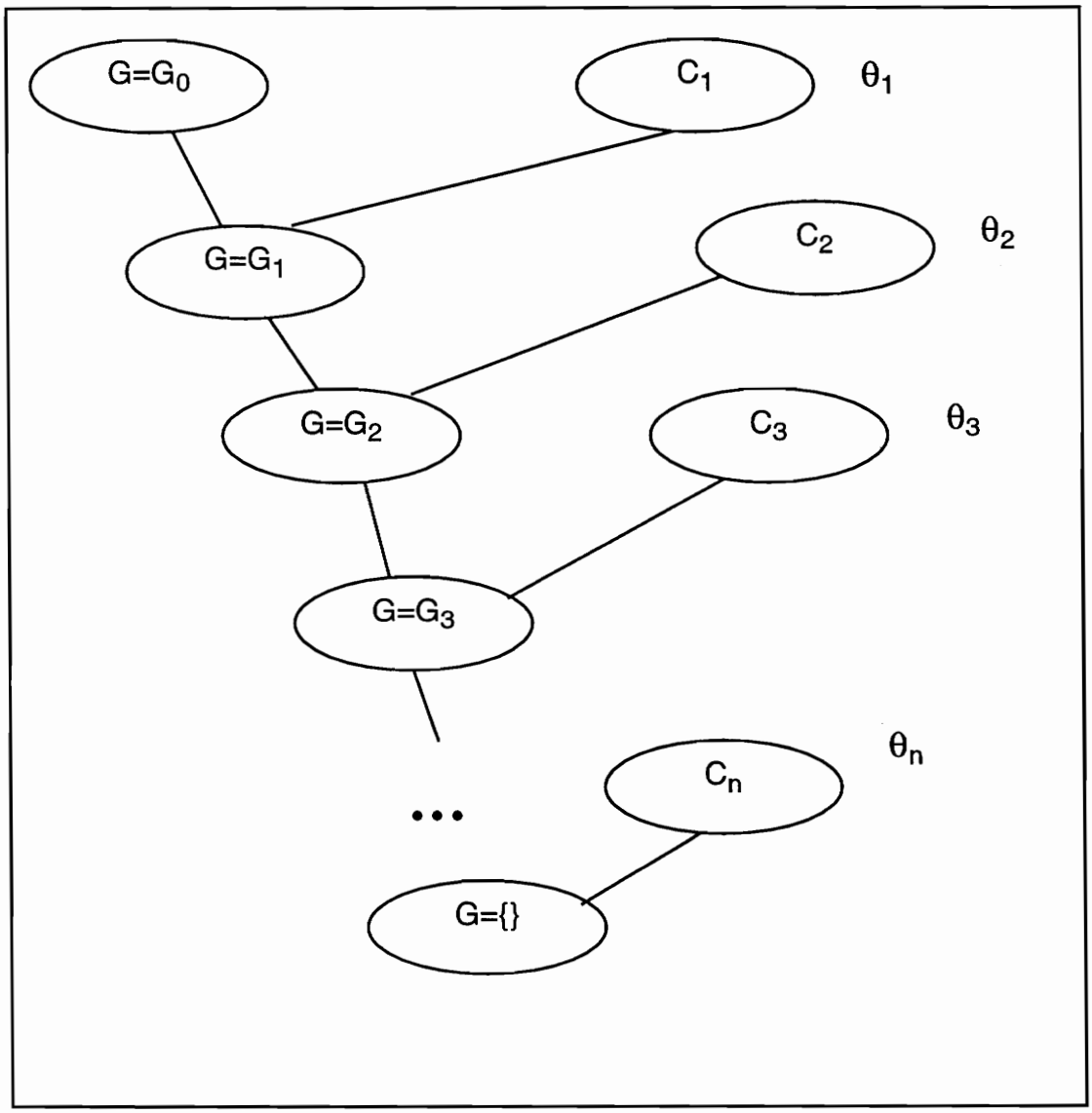

Figure 15 General Linear Refutation Tree

A linear refutation tree is built from the top node, which represents the goal, and finishing at the root node with a null symbol {} signifying the disproving of the original assertion. In the figure, The G_i consist of negated literals only and represent derived goals or sub-goals. The C_i are clauses from R $\cup$ EDB or variants of R $\cup$ EDB. The θ_i are the substitutions at each level of the tree.

SLD-resolution uses *selection* functions. The most widely used selection function is the one which associates each goal with its left-most literal. Another selection function is choosing the literal containing the maximum number of constants. If there is a tier, then the selection uses the left-most literal.

A *resolvent* is produced by a goal (or a group of goals) with either a rule or a fact. The resolvent itself is a goal (possibly compound) or the null clause.

SLD-resolution

SLD-resolution is a method of resolution via refutation. Let *S* be a set of Datalog clauses, let *G* be a Datalog goal, and let *s* be a selection function.

An SLD-refutation of S $\cup$ {G} via *s* consists of a finite sequence of goals G = G_0,..., $G_n = G_{\{\}}$, a sequence C_1,..., C_n of clauses, and a sequence of substitutions θ_1,..., θ_n such that the following conditions are satisfied:

- Each clause C_i, for $1 \leq i \leq n$ is either a clause of S or a variant of a clause of S.
- Each goal G_i, for $1 \leq i \leq n$ is a resolvent of the clauses G_{i-1} and C_i according to selection function *s*.

- For $1 \leq i \leq n$, θ_i is the substitution used in the resolution of G_{i-1} and C_i. The formal name of the substitution is *Most General Unifier* (MGU) in the context of logical programming [Nils80].

Figure 16 gives an example of solving a query via SLD-resolution. The following lists the rules in the IDB, facts in the EDB, and the query, respectively.

IDB

R1: p (X, Z):- p (X, Y), p (Y, Z)

R2: p (X, Y):- p (Y, X)

EDB

p (a, b)

p (b, c)

p (c, d)

p (d, e)

Query

Q (X):- p (a, X) and thus the goal is $\neg$ p (a, X)

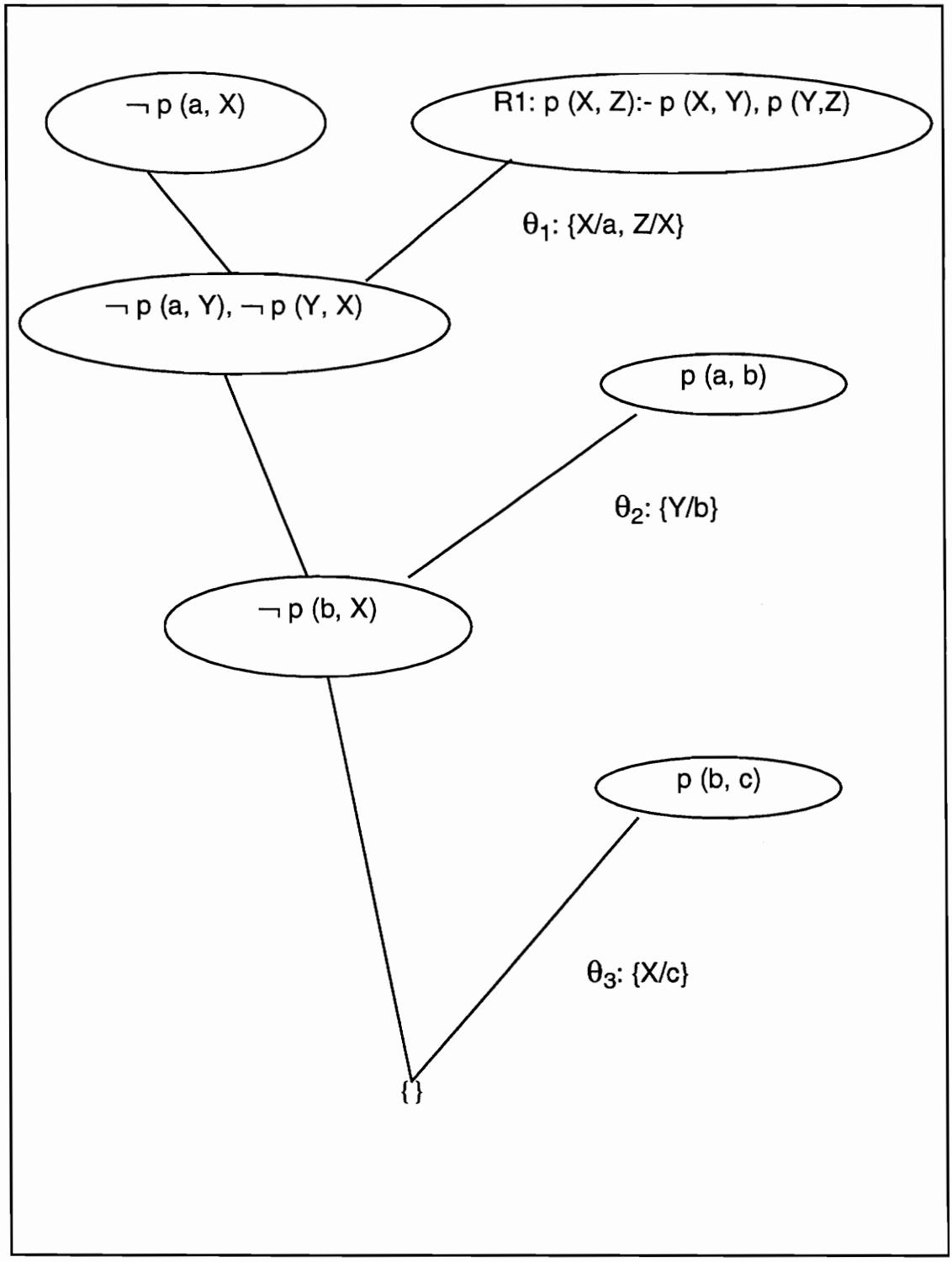

Figure 16 Solving a Query via SLD-Resolution

The example shows that the refutation produced a counter-example to the goal. This counter-example is p (a, c), one of the solutions to the query.

6.1.2 Bottom-Up Evaluation

A *bottom-up* evaluation algorithm starts the evaluation by considering the facts at the bottom of the proof tree. The algorithm then works upward by first producing all facts which can be inferred in one step from the bottom clauses; next, producing all facts which can be computed in one step from the bottom clauses or from facts which were inferred in one step from the current level of the tree; and so on. Eventually the algorithm produces all the facts which can be computed from the rules and the new and existing facts. The final step is to produce the answer to the query by discarding all generated irrelevant facts.

Bottom-up algorithms are usually easier to implement than *top-down* algorithms. However, *bottom-up* algorithms usually compute a lot of useless results because the algorithms do not have the knowledge, in terms of the query they are evaluating.

There are two main algorithms in the family of *bottom-up* evaluation methods. They are *Naive evaluation* and *Semi-Naive evaluation* algorithms. These two algorithms are introduced in the following subsections.

6.1.2.1 Naive Algorithm

Naive evaluation is a process of inference, inferring new facts based on deductive rules and database facts. Naive evaluation is not a query optimization technique, however, it is a basic bottom-up method for evaluating a query. It is important to understand this method in order to understand some more efficient evaluation methods such as the Semi-Naive evaluation method.

Before the introduction of the Naive method, it is necessary to introduce briefly a general inference rule. Consider a deductive rule R of the form L_0:- L_1,..., L_n and a set F of ground facts F_1,..., F_n. If a substitution θ exists such that, for each $1 \leq i \leq n$, $L_i \theta = F_i$, then from the rule R and from the facts F_1,... F_n, the fact $L_0\theta$ can be inferred in one step. Note that $L_0\theta$ could either be a new fact or an existing fact. For example:

from R: p (X, Z):- p (X, Y), p (Y, Z) and

F: {p (a, b)} and {p (b, c)}

a new fact {p (a, c)} can be inferred in one step. However, in the following example, the inferring process produces an existing fact:

assume R: p (X, Y): - p (Y, X) and

F: {p (a, a)}

then the inferring process will produce {p (a, a)}, which is an existing fact.

Figure 17 presents the Naive evaluation algorithm. In the algorithm, the function *infer* takes as input a finite set of facts F and a finite set of rules R, and produces the set of all facts which can be inferred in one step from F and R. The algorithm terminates when the *infer* function no longer produces new facts. In [CGT90], there is a proof that the algorithm guarantees termination.

```
Input:    a finite set F of Datalog database facts
          a finite set R of Datalog database rules
          a query
Output:   a set of facts which can be inferred from F and R

begin
    old = NULL;
    new = NULL;
    repeat
          old = new;
          new = new ∪ infer (R, F+ new);
    until new = old;
    evaluate the query by using the facts in new;
end.
```

Figure 17 Naive Evaluation Algorithm

Now let's use the famous *ancestor* query to demonstrate how the Naive algorithm can be used to evaluate a query. Consider a database with the following rules (IDB) and facts (EDB):

IDB

R1: ancestor (X, Y):- parent (X,Y).

R2: ancestor (X,Y):- ancestor (X, Z), parent (Z, Y).

EDB

parent (a, aa).
parent (a, ab).
parent (aa, aaa).
parent (aa, aab).
parent (aaa, aaaa).
parent (c, ca).

The IDB states that if Y is a parent of X, then Y is an ancestor of X. Also, if Y is a parent of Z, and Z is an ancestor of X, then Y is an ancestor of X.

The EDB states that there exist the following facts: *aa* is a parent of *a*, *ab* is a parent of *a*,..., and *ca* is a parent of *c*.

The query is *Q (aa, W)* - to find all ancestors of *aa*. Figure 18 shows the output of each iteration for evaluating this query.

Iteration 1:

Infer generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}

Iteration 2:

Infer generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa)}

Iteration 3:

Infer generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

Iteration 4:

Infer generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

Since this iteration did not produce any new fact, the algorithm exits the loop.

Now query *Q (aa, W)* is evaluated by using the constant *aa* against the facts produced by the algorithm, and the answer set is {(aa, aaa), (aa, aaaa), (aa, aab)}, i.e., the ancestors of *aa* are *aaa*, *aaaa*, and *aab*.

Figure 18 Query Evaluation via Naive Algorithm

Note that the Naive algorithm is inefficient, primarily due to two reasons:

- Redundant work:

The input to the inference process is the set F and the entire set of facts, *new*, produced from the previous iteration. Therefore, any inference process performed at one iteration will also be performed at each subsequent iteration. For example, the entire computation of *Iteration 2* is duplicated in *iteration 3.*

- Computation of irrelevant facts:

The algorithm does not use the constants in the query. As a consequence, the entire set of facts is used for the computation whether a fact is relevant to the query or not. For example, the fact *ancestor (a, aa)* is irrelevant to the query; nevertheless, it is derived and used in the computation.

6.1.2.2 Semi-Naive Algorithm

The previous section pointed out that one of the reasons for the inefficiency of the Naive evaluation was that the algorithm performed redundant computations, and the redundant computations were caused by the redundant input of the existing facts produced by the previous iteration. For example, in Figure 18, *Iteration 2* generated new facts {(a, aaa), (a, aab), (aa, aaaa)} and existing facts {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}. Both new facts as well as existing facts were used as input, *new,* to the *iteration 3.* Based on these existing facts, the computation done by *iteration 3* was the same computation done by *Iteration 2* on the same facts. Therefore, that portion of the *iteration 3* computation was redundant.

The Semi-Naive algorithm uses the same bottom-up approach in its evaluation as the Naive algorithm. The improvement of the Semi-Naive algorithm over the Naive algorithm is that the Semi-Naive algorithm eliminates the redundant work of the Naive algorithm in processing recursive queries. The performance improvement of Semi-Naive algorithm over Naive algorithm is by orders of magnitude [BR89]. The Semi-Naive algorithm achieves this improvement via the following strategy:

> At each iteration, rules are applied only to the new facts (not the existing facts) generated by the previous iteration.

Figure 19 presents the Semi-Naive evaluation algorithm.

```
Input:    a finite set F of Datalog database facts
          a finite set R of Datalog database rules
          the query
Output:   a set of facts which can be inferred from F and R

begin
    old = NULL;
    new = NULL;
    repeat
          new_Δ = new - old;
          old = new;
          new = new ∪ infer (R, F+ new_Δ);
    until new = old;
    evaluate the query by using the facts in new;
end.
```

Figure 19 Semi-Naive Evaluation Algorithm

Notice that in the algorithm, the differential new_{Δ} is used instead of *new* as the input to *infer.* Therefore infer will perform its computations with the smaller set new_{Δ} instead of the entire set of *new.* As a result, the algorithm eliminates redundant computation. Note that this algorithm works for linear recursive rules only, because in the case of linear recursive rules, the differential is simply *new - old* [BR89]. Figure 20 shows the output of each iteration for evaluating the same *ancestor* query of Section 6.1.2.1 via the Semi-Naive algorithm.

Iteration 1:

Infer generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}

Iteration 2:

Infer generated:

{(a, aaa), (a, aab), (aa, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa)}

Iteration 3:

Infer generated:

{(a, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

Iteration 4:

Infer generated:

NULL

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca),
(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

Since this iteration did not produce any new fact, the algorithm exits the loop.

Now query *Q (aa, W)* is evaluated by using the constant *aa* with the facts produced from the algorithm, and the answer set is {(aa, aaa), (aa, aaaa), (aa, aab)}, i.e., the ancestors of *aa* are *aaa*, *aaaa*, and *aab*.

Figure 20 Query Evaluation via Semi-Naive Algorithm

For non-linear rules, differentials of higher order can be used to calculate new_{Δ}. However, the expressions become quite cumbersome [CGT90]. The alternative is to translate the non-linear rule into a set of linear rules. But the Semi-Naive algorithm for the translated set of linear rules will produce less computation savings than the linear Semi-Naive algorithm. Since the most common cases involve only linear rules [BR89, CGT90], and since the non-linear rules are not considered by this research, the non-linear Semi-Naive algorithm is not presented here.

6.1.2.3 Magic-Sets Rewrite Algorithm

As noted in Section 6.1.2.1 there are two main reasons that cause the inefficiency of the Naive evaluation: redundant computation and irrelevant facts computation. The Semi-Naive algorithm eliminates the duplicated computation; however, it still uses the entire set of facts in the computation. Thus the algorithm produces facts which are irrelevant to the query. For example, the fact *ancestor* (a, aa) is irrelevant to the query *Q (aa, W)*; nevertheless, it is generated and used in the computation. The facts produced by the computation are irrelevant to the query, and thus the computations are wasted effort. According to [BR89], the performance improvement of Magic-Sets rewriting with Semi-Naive evaluation over pure Semi-Naive evaluation is of orders of magnitude.

The Magic-Sets Rewrite algorithm tries to reduce the number of irrelevant facts by rewriting the rules based on the constants provided in the query. As a result, the evaluation using the rewritten rules will greatly reduce the wasted effort.

Before presenting the algorithm, some definitions must be introduced. The two relevant definitions are: *Distinguished Argument* and *Reachable adorned Rules.*

Distinguished Argument

An argument of a sub-goal in rule r is *distinguished* if one of the following conditions holds:

- it is a constant,
- it is bound by the adornment (corresponding to a bound argument in the head), or
- it appears in an EDB predicate occurrence that has a distinguished argument.

Thus, arguments in an EDB predicate are either all *distinguished* or all *not distinguished.*

Reachable Adorned Rules

An adorned rule is *reachable* for the goal iff:

- it is the adorned rule corresponding to the goal rule, with all the *left-hand-side* predicate arguments free, or
- its head predicate appears, with the same adornment, in the *right-hand-side* of a reachable rule.

Figure 21 presents the Magic-Sets Rewrite algorithm.

$P^{magic} = P^{A}$; // P^{A} is a set of adorned rules.

FOR each adorned rule r, and FOR each occurrence of an intensional predicate p in the RHS Of r DO

BEGIN // loop 1

Generate on magic rule in the following way:

a) Delete all other occurrences of IDB predicates in the RHS;

b) Replace the name of p in this occurrence with magic_r_p^{a}_i where a is the adornment of p in that occurrence and i is the occurrence number;

c) Delete all non-distinguished variables of this occurrence of p, thus possibly obtaining a predicate with fewer arguments;

d) Delete all non-distinguished EDB predicates in r;

e) Replace the name of the head predicate p' with magic_p' $^{a'}$, where a' is the adornment of p';

f) Delete all non-distinguished variables of p';

g) Exchange the places of the head magic predicate and the body magic predicate.

Add this rule to P^{magic}.

END

FOR each adorned rule r in the original program DO

BEGIN // loop 2

FOR each occurrence of an intensional predicate p in the RHS of r DO

BEGIN

Add to the RHS the predicate magic_r_p^{a}_i (X)

where a is the adornment of the occurrence of p, i is the occurrence number, X is the list of distinguished arguments in this occurrence.

IF p is not the head predicate THEN

insert the magic predicate just before that occurrence;

ELSE

insert the magic predicate at the beginning of the rule body

END

Replace r with its modified version in P^{magic};

END

FOR each IDB predicate p and FOR each adornment DO

BEGIN // loop 3

FOR each adorned ruler, and FOR each occurrence of p in the RHS Of r DO

BEGIN

Add the rule: magic_p^{a} (X):- magic_r_p^{a}_i (X), where i is the occurrence of p, a is its adornment, and X is the list of its distinguished arguments.

END

Add this rule to p^{magic}.

END.

Figure 21 Magic-Sets Rewrite Algorithm

Let's again consider the *ancestor* query and the database used in Section 6.1.2.1. The rules are:

R1: ancestor (X, Y):- parent (X,Y).
R2: ancestor (X,Y):- ancestor (X, Z), parent (Z, Y).

and the query is Q (aa, W).

After the Magic-Sets rewriting, the rules became:

R1: ancestor (X, Y):- parent (X,Y).
R2: ancestor (aa, Y):- ancestor (aa, Z), parent (Z, Y).

As can be seen immediately, only the facts related to *aa* in R2 will be computed. Thus the Magic-Sets algorithm achieves optimization by limiting the facts used in the computation. All of the steps in this rewriting are presented in Appendix E. Figure 22 shows the iterations of Semi-Naive evaluation of the query *Q (aa, W)*, using the above rewritten rules.

Iteration 1:

New facts generated:

{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)}.

Iteration 2:

New facts generated:

{(aa, aaaa)}

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (aa, aaaa)}

Iteration 3:

New facts generated:

NULL

new = {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (aa, aaaa)}

Since this iteration did not produce any new fact, the algorithm exits the loop.

Now query *Q (aa, W)* is evaluated by using the constant *aa* with the facts produced from the algorithm, and the answer set is {(aa, aaa), (aa, aaaa), (aa, aab)}, i.e, the ancestors of *aa* are *aaa*, *aaaa*, and *aab*.

Figure 22 Magic-Sets Rewriting with Semi-Naive Evaluation

6.2 Some Deductive Object-Oriented Query Optimization Systems

This section examines some deductive object-oriented query optimization systems, in particular, the OOA system [Gard94] and the O-Telos system [JS94, JGJS95]. The O-Telos system will be presented in detail because it is one of the

most complete systems in terms of its data model and query language. Also it has been implemented in a prototype called *ConceptBase*. Versions of ConceptBase have been distributed for research experiments since early 1988. The 1993 ConceptBase V3.2 has been installed at more than one hundred sites worldwide [JGJS95].

6.2.1 The OOA System

Gardarin proposed an object-oriented algebra, OOA [Gard94]. OOA is an extended relational algebra designed to support methods and other object-oriented features. In addition, OOA supports deductive model features such as recursion and quantifier. The algebra supports the following operations:

1. Restriction: Restriction takes a collection as input and produces a collection of the same type whose objects satisfy a (possibly complex) condition.
2. Projection: Projection produces a new collection from a given one by computing the expressions of source attributes as target attributes.
3. Join: Join is defined as a Cartesian product of two collections, followed by a restriction.
4. Set operations: union, difference, and intersection.

Collections are defined as set, list, bag, array, etc. familiar groups.

OOA implements the recursive operations via algebraic iteration. The algebra implements the *fixpoint* computation. The fixpoint computation simply produces the saturation of a collection computed recursively by an algebraic expression [Ullm88].

The system supports both syntactic and semantic optimization. The syntactic optimization is performed via syntactic transformations, which include the following:

- Composition and Decomposition of Operations: A complex operation can be decomposed into more basic operations such as *restriction* and *projection*. On the other hand, simple operations can be composed into a more complex operation. Composition and Decomposition generate various execution plans. Heuristics such as grouping the operations which operate on the same class may be used to reduce the number of plans.

- Permutation of Operations: Permutation rules use constraints on the classes to reduce the relevant facts. Permutation rules are heuristic and do not guarantee a better access plan. One example of permutation rules is performing *selections* before *joins*.

- Common Sub-Query Isolation: Common sub-queries should be replaced by a single sub-query with a unique output. This avoids duplicated computation on the same sub-queries.

- Fixpoint Reduction: In the case of recursive predicates, the permutation between operators can not be done easily. OOA proposes to use algorithms such as Magic-Sets rewriting to reduce the number of relevant facts.

The OOA system emphasizes the use of semantic optimization in addition to syntactic optimization. Gardarin notes that the rich semantics of object-oriented database systems made semantic optimization important. Semantic optimization is achieved via a semantic transformation. The system proposes three main semantic transformations:

- Abstract Data Type Function Rewriting: Certain queries are expressed over well known abstract data types such as sets, lists, and arrays. These structures have basic algebraic operations such as union, intersection, etc., as well as privileged predicates such as equality and membership associated with them. The rewriting can thus be based on the relationships of the operations. For example, an intersection operation can be expressed via a difference operation.

- Simplification via Integrity Constraint: The basic idea of using integrity constraint to optimize a query is well known. For example, if a constraint states that all cars are made in the U.S., then a query requesting information about cars made in Japan will be given a null answer without even going through the computation. However, choosing appropriate integrity constraints that simplify query processing is a difficult task. [Gard94] does not provide any rule which may be used to choose the appropriate constraints.

- Operator Property Based Transformation: This transformation concerns the semantics of operators. In object-oriented systems, operators can be overloaded. Thus, it is useful to be able to define rules using the properties of operators in query optimization. [Gard94] uses the transitivity of

equality as an example to demonstrate this concept. One example is that the system can define a rule such as "a = b and b = c is equivalent to a = b and a =c." This will enable the optimizer to generate an alternative access plan.

6.2.2 O-Telos and ConceptBase

O-Telos is a deductive object-oriented data model used by ConceptBase. ConceptBase is a prototype deductive object-oriented database developed in Germany [JGJS95]. The intended applications of that research range from "uniform access to heterogeneous data sources" to the "integration of heterogeneous information services in networked enterprises."

6.2.2.1 Data Model

O-Telos is a deductive object-oriented data model. The model defines an extensional deductive object database (DOB) as a triple:

DOB = (OB, IDB, IC), where

OB is the extensional object base traditionally referred to as EDB, IDB is a set of rules, and IC is a set of integrity constraints. It is required that (OB, IDB, IC) be consistent, i.e., OB $\cup$ IDB $\models$ IC. The formulas in IDB $\cup$ IC have to be range-restricted, which is a widely accepted sufficient condition for domain independence. The set IDB is stratified to ensure the unique perfect model defined in [CGT90]. Stratification simply means computing a predicate before using its negation.

The model is composed of two major parts:

- object structures (OB), and
- deductive rules (IDB) and integrity constraints (IC).

The following subsections describe the two parts.

6.2.2.1.1 Object Structures

The semantics of the object structure is defined by specifying the extensional database structures and the predefined axioms. An extensional object base (OB) is a finite subset

$OB \subseteq \{P\ (o, x, r, y)\ |\ o, x, y \in ID, r \in LAB\}$, where

P is the only base literal in the OB, ID is a set of identifiers, and LAB is a set of labels. The elements of OB are called objects with identifier *o*, source components *x*, destination components *y*, and label *r*.

The model captures the semantics of the object-oriented model as follows:

• Individual:	P (o, o, r, o)	Individual object
• Instantiation:	P (o, x, in, c)	x is in class c - IN
• Specialization:	P (o, c, isa, d)	c is a subclass of d - ISA
• Attribution:	P (o, x, r, y)	relationships - aggregation / association

The semantics behind an object *P (o, x, r, y)* is that there is a relation *r* between objects *x* and *y*. The relationship is itself an object with identifier *o*. Instantiation and specialization are actually the two special cases of *P (o, x, r, y)* with *r* being replaced by *in* and *isa* respectively. O-Telos uses the convention that the object identifier (OID) of an object with name *n* is written as *#n*. For example, the OID for *John* is written as *#John*.

An OB can be represented as a structured semantic net. *Individuals* are represented as nodes with name *r*, *instantiation* is represented as dotted links, and *specialization* is represented as shaded links. All other *relationships* are represented as direct links. Figure 23 shows a sample OB and Figure 24 gives object structure examples corresponding to the class and data definitions in Figure 23.

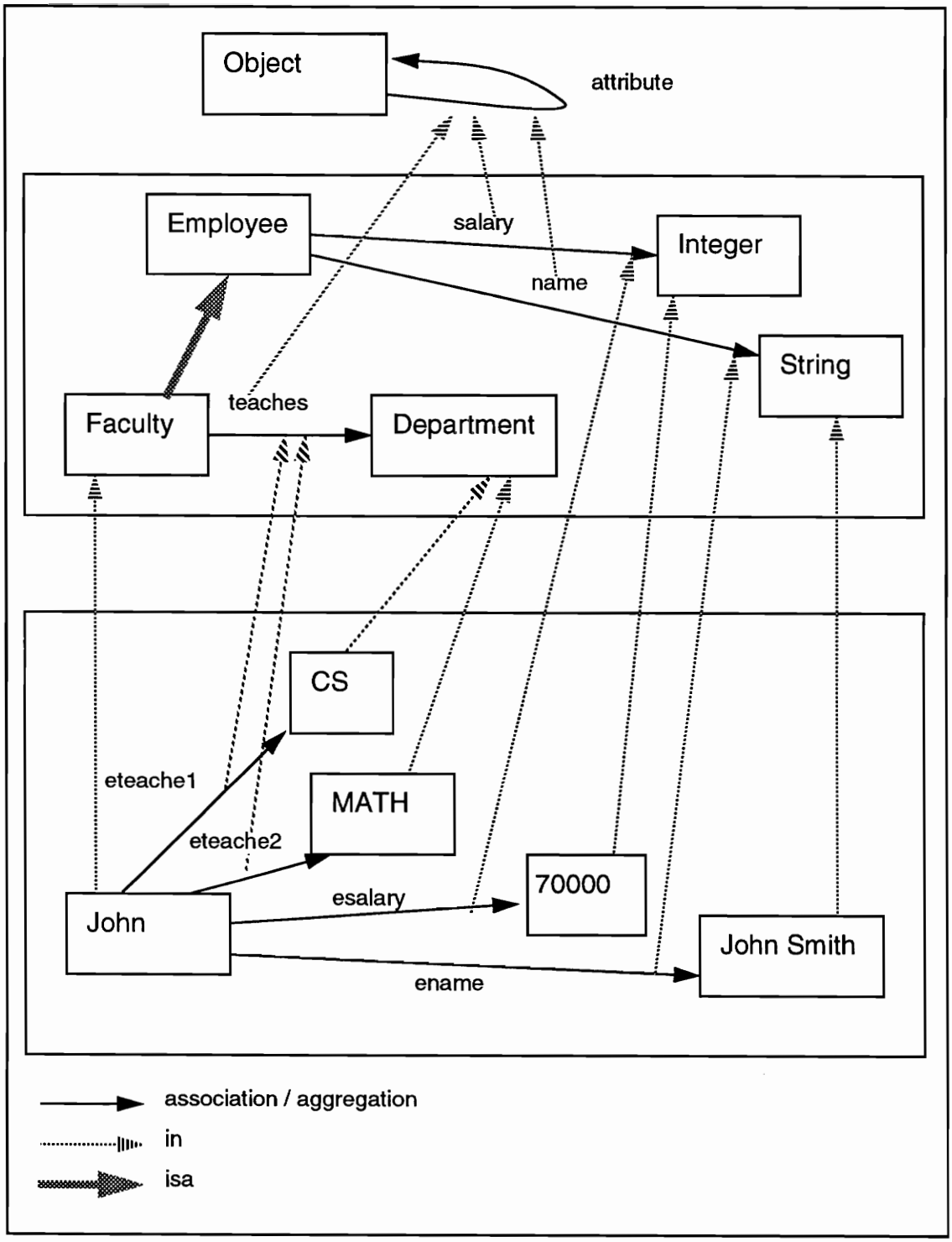

Figure 23 O-Telos Sample Object Database

In Figure 23 the OB has two layers. The top layer is called the Simple-Class layer, which corresponds to the schema in traditional databases. Employee, Faculty, Department, Integer, and String are all classes. A class is both an object (ISA) and an instance of object (IN). There is no restriction on the number of layers. The schema can be instances of a set of metaclasses, the set of metaclasses can be instances of yet another set of metaclasses, and so on. The top layer of an object base is predefined by the *Object*, which contains all objects in the OB.

The lower layer is called the Token layer which contains all stored objects, the actual data in the traditional databases. The token layer of this OB contains an object with the label, *John*, which is an instance of the object *Faculty*. John teaches in two departments, CS and MATH, which are instances of *Department*. The object contains "John Smith" which is an instance of *String*, and 70000 is an instance of *Integer*.

Simple-Class Definition:

P (#Employee, #Employee, Employee, #Employee) (L1)
P (#name, #Employee, name, #String) (L2)
P (#salary, #Employee, salary, #Integer) (L3)

Token Definition:

P (#John, #John, John, #John) (L4)
P (#in1, #John, In, #Faculty) (L5)
P (#ename, #John, ename, "John Smith") (L6)
P (#in2, #ename, In, #name) (L7)
P (#esalary, #John, esalary, 70000) (L8)
P (#in3, #esalary, In, #salary) (L9)

Figure 24 O-Telos Sample Object Structures

Figure 24 shows a schema definition and a token definition corresponding to the *Employee* class and employee *John* instance. The following explains a few lines in Figure 24.

L1 defines an object with label *Employee* and OID *#Employee*. It is in the format of *Individual*.

L2 defines a relationship between class *#Employee* and class *#String*. The relationship is itself an object with OID *#name* and label *name*.

L4 defines an object with label *John* and OID *#John*. It is in the format of *Individual*.

L6 defines a relationship between object *John* and *"John Smith"*. The relationship has an OID of *#ename* and label *ename*.

L7 states that *#ename* of L6 is an instantiation of *#name*. The instantiation has a label *In* and OID *#in2*. *In* is a special case of *relationship*.

An OB can also be represented in a frame-based notation, which is based solely on object labels. This representation is similar to the regular class representation. Figure 25 shows a few examples of the classes from Figure 23.

```
Employee In SimpleClass with
    attribute
        name: String;
        salary: Integer;
end Employee.

Faculty In SimpleClass isA Employee with
    attribute
        teaches: Department;
end Faculty.

John In Faculty with
    name      ename: "John Smith";
    salary    esalary: 70000;
    teaches   eteach1: CS;
              eteach2: MATH;
end John.
```

Figure 25 O-Telos Sample Class Definitions

O-Telos uses predefined deductive rules and integrity constraints to implement object-oriented features present in most object-oriented data models. These rules and constraints are referred to as axioms. The following presents a few typical features such as object identity, instantiation, specialization/generalization, aggregation and typing.

Object Identity:

$\forall$ o, x_1, r_1, y_1, x_2, r_2, y_2, P (o, x_1, r_1, y_1) $\wedge$ P (o, x_2, r_2, y_2) $\Rightarrow$

$$(x_1 = x_2) \wedge (r_1 = r_2) \wedge (y_1 = y_2) \qquad (A_1)$$

The axiom states that no two objects in the OB have the same identifiers.

Instantiation:

$$\forall o, x, c\ P\ (o, x, in, c) \Rightarrow In\ (x, c) \qquad (A_2)$$

This axiom induces instantiation - *In (x, c). In (x, c)* can then be used to form deductive rules and integrity constraints. The axiom states that if any object *x* (with OID *o*) is an instance of *c*, then the same *x* (without referencing to *o*) is an instance of *c*.

Specialization:

$$\forall o, c, d\ P\ (o, c, isa, d) \Rightarrow Isa\ (c, d) \qquad (A_3)$$

This axiom induces specialization - *Isa (c, d). Isa (c, d)* can then be used to form deductive rules and integrity constraints. The axiom states that for any object *c* (with OID o) *Isa d*, then the same *c* (without referencing to o) *Isa d*.

Aggregation / Attribution:

$$\forall o, x, r, y, p, c, m, d\ P\ (o, x, r, y) \wedge P\ (p, c, m, d) \wedge In\ (o, p) \Rightarrow A\ (x, m, y) \qquad (A_4)$$

This axiom induces aggregation - A (x, m, y). A (x, m, y) can then be used to form deductive rules and integrity constraints. The axiom states that if o is an instance of *p*, then *r* must be an instance of *m*. Thus the attribute is a relation between *x* and *y* with a mapping *m*. For example, P (#esalary, #John, esalary, 70000) $\wedge$ P (#salary, #Employee, salary, #Integer) induces A (#John, salary, 70000). The axiom provides a single literal A (x, m, y) for all attribute accesses from an object *x* to its attribute value *y*. In object-oriented notation this is written as *x.m = y*.

Inheritance / Class Hierarchy:

$\forall$ x, c, d In (x, c) $\wedge$ Isa (c, d) $\Rightarrow$ In (x, d) (A_5)

This axiom states that if x is an instance of c and c *Isa* d then x is an instance of d as well.

Typing:

$\forall$ o, x, r, y, p P (o, x, r, y) $\wedge$ In (o, p) $\Rightarrow$
$\exists$ c, m, d P (p, c, m, d) $\wedge$ In (x, c) $\wedge$ In (y, d) (A_6)

This axiom states that the attributes of an object must be correctly typed according to the attribute definitions of its classes.

6.2.2.1.2 Deductive Rules and Integrity Constraints

Deductive rules and integrity constraints are range-restricted first order formulas over the three literals *In*, *Isa*, and *A*. The object structure and the axioms offer only one base relation *P* and three deduced relations *In*, *Isa*, *A*. That number is surely not satisfactory, i.e., the entire database has only one predicate P. In order to extend the deduction machine, a restricted interpretation of deductive rules and integrity constraints is adopted.

An object database with only axioms A_2 to A_4 as deductive rules delivers ground facts for the three predicates. Note that these axioms are range-restricted and stratified. For a ground fact A (x, m, y), the closure axiom applied to A_4 guarantees the existence of an object P (p, c, m, d) to which an attribute of *x* was instantiated. This can be extended to a fact A.p(x, y) for each such *p*, and *p* can be

any user defined object. Thus the database can be represented by a single predicate *P* as well as many predicates *p*. Similarly, a fact In.c(x) for each ground fact In (x, c) can be derived.

The above modifications extend the number of literals by all class and attribute identifiers. The new deductive rules and integrity constraints now contain A.p, In.c and Isa literals.

6.2.2.2 Queries and Query Language

O-Telos views queries as class as well as deductive rules. The O-Telos query language is based on the Telos language [MBJK90]. Telos represents one of the earliest attempts to integrate deductive and object-oriented data models.

The O-Telos query language is a formal query language. O-Telos is equivalent in expressiveness to Datalog, with stratified negation and perfect model semantics. Also, an object algebra, COBRA, was defined for O-Telos [Theo92].

The O-Telos query language is a language with predicates based on behavior. The language allows the creation of new objects, and therefore it is an object-creating language. O-Telos is a non-restrictive language. The language supports quantifiers and recursive queries.

From a structural point of view, O-Telos is a very powerful semantic modeling language whose distinguishing features include attributes as full-fledged objects and a potentially infinite hierarchy of classification called metaclasses.

6.2.2.2.1 Query as Classes and Deductive Rules

In O-Telos, a query is expressed in the frame notation which is basically the same as a class definition. A special class defined by the system, *QueryClass*, contains all possible queries which themselves are classes.

Following the instantiation and classification principles, instances of a query class are answers to the query. Deductive rules and integrity constraints specify necessary and sufficient membership conditions for answer instances in query classes.

Query classes can have superclasses. Superclasses and subclasses are connected via *Isa* links. These superclasses restrict the set of possible answer instances to the common instances of the superclasses of the query class.

Query classes may have two different types of attributes. The first type is inherited from one of the superclasses. If such an attribute is specified explicitly in a query class description, the answer instances are given back with value instantiation of this attribute, which is similar to the *projection* operation in relational algebra.

The second type of attribute is one whose instantiation value by an answer instance is computed during the query evaluation process. This means that a relation between the answer instance and the computed attribute value is not necessarily stored in the database or deducible by a stored deduction rule. In order to express relationship, *typed first-order logic* expressions are used in the query classes.

The semantics of query classes is twofold. First, a query class is a class object which has some attributes. Second, there is a mapping from the query class to a deductive rule that defines which objects are the answer to the query. Consider the generic query class:

QueryClass Q isA C_1,..., C_k with
  attribute
    a_1: S_1;...; a_m: S_m;
    b_1: T_1;...; b_n: T_n;
  constraint
    c: $<formula text> $
end

where a_1,... a_m are attributes which are refined (specialized) from existing attributes with the same labels in classes C_1,... C_m. The attributes b_1,..., b_n are additional properties of Q. Let ψ be the first-order formula represented by < formula text>. Then the query rule corresponding to the query class is:

$$\forall x, y_1,..., y_m, z_1,..., z_n \; In.\#C_1(x) \wedge ... \wedge In.\#C_k(x) \wedge$$
$$In.\#S_1(y_1) \wedge A.\#a_1(x, y_1) \wedge ... \wedge In.\#S_m(y_m) \wedge A.\#a_m(x, y_m) \wedge$$
$$In.\#T_1(z_1) \wedge ... \wedge In.\#T_n(z_n) \wedge \psi \Rightarrow Q(x, y_1,..., y_m, z_1,..., z_m) \qquad (1)$$

The first argument x of Q is called the *answer variable* of the query. The other arguments are called *query attributes*. The constants $\#C_i$, $\#S_i$, $\#T_i$ are the object identifiers of the labels C_i, S_i, T_i. Constants $\#a_1$,..., $\#a_m$ are the identifiers of the attributes labeled a_1,..., a_m. Formula (1) shows that the variables in Q (x, y_1,..., y_m, z_1,..., z_m) are bound to objects in the object database. There is no creation of object identifiers for answers to queries.

Since query classes may be superclasses or attribute value classes of other queries, class membership to queries is defined by a second formula:

$$\forall x, y_1,..., y_m, z_1,..., z_n\ Q\ (x, y_1,..., y_m, z_1,..., z_m) \Rightarrow In.\#Q(x) \qquad (2)$$

The evaluation of this rule leads to a set of ground instances of the literal Q which can be used to build the answer instances of the query class Q.

6.2.2.2.2 Object Algebra

The object algebra for O-Telos is COBRA [Theo92]. COBRA is based on the relational algebra and extended with object-oriented features. In that respect, COBRA is similar to the object algebra defined for SOSYS.

COBRA separates algebraic equations generated from user-defined rules and queries, and the basic equations allow access to the stored extensional database. This approach allows easy integration of alternative storages.

In addition to the basic equations, equations for instantiation and specification are defined. These equations are basically the algebra implementation of the *In*, *Isa*, and *A* predicates defined at the logic level.

Works related to COBRA have been published in German. Since there has been no English translation for COBRA works so far, further discussion will not be provided here.

6.2.2.3 Query Processing Methodology

O-Telos uses a comprehensive query processing methodology. The comprehensiveness is compatible with SOSYS in that processing involves both rule (calculus) and algebra optimization. Figure 26 depicts the O-Telos query processing methodology.

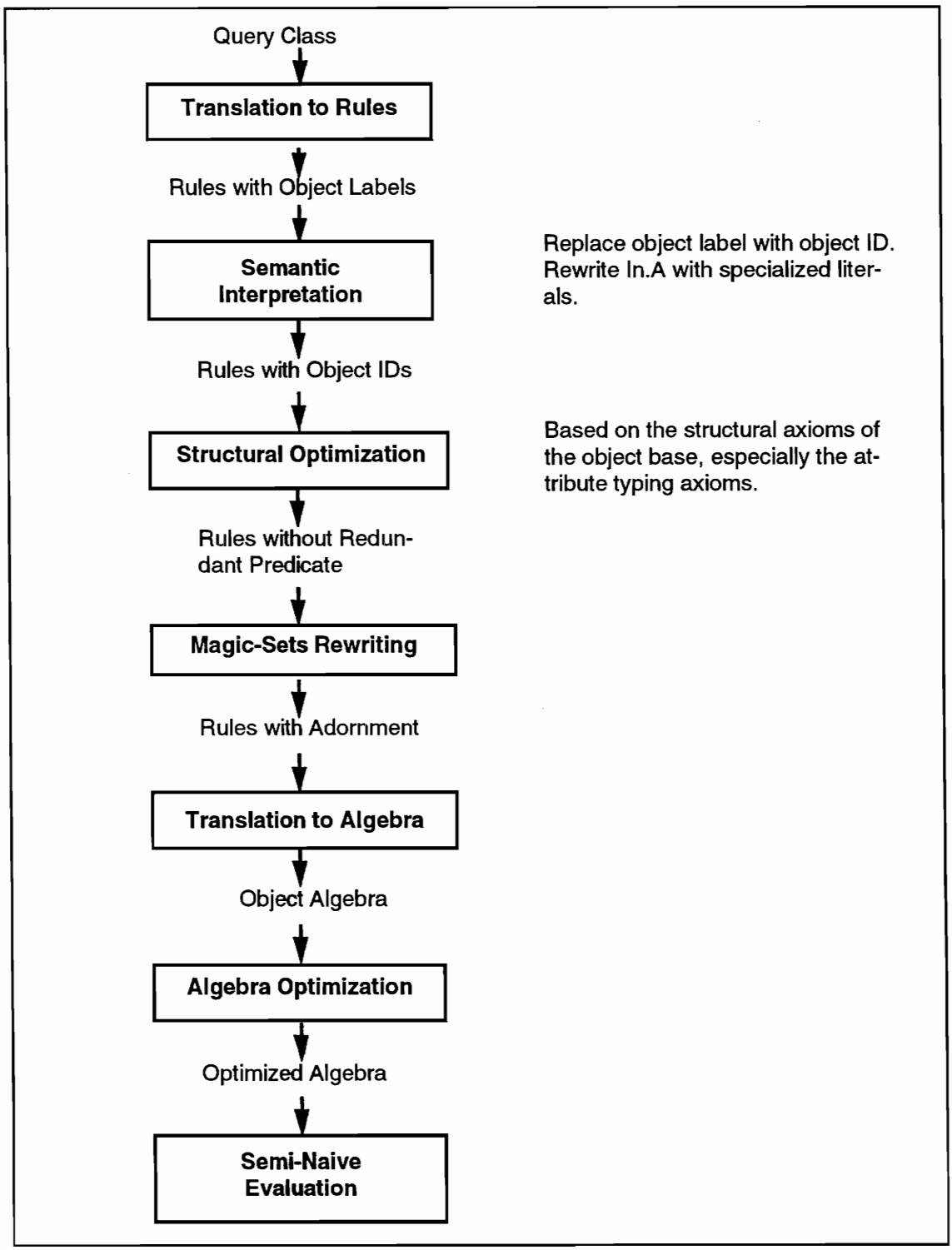

Figure 26 O-Telos Query Processing Methodology

The methodology involves the following basic steps. First, a query class is translated into deductive rules. These rules will go through the semantic interpretation and structural optimization processes. These processes will produce an O-Telos internal representation of the deductive rules in a format similar to Datalog rules. For referencing purposes, these rules are called *i-rules* which corresponding to the output from the structural optimization stage in Figure 26. Any optimization strategy applicable to Datalog rules can then be used to optimize the i-rules. The optimized i-rules are then translated into the equations of COBRA. These equations are further optimized via algebra optimization. Finally, the optimized algebra equations are evaluated via Semi-Naive evaluation.

6.2.2.3.1 Structural Optimization

Structural optimization in O-Telos is via the exploration of the structural axioms of the object database. These axioms are theorems, and they are true in any consistent deductive object database (OB, IDB, IC). Therefore, they can always be used to simplify and optimize queries. This section gives an example of query optimization via axioms A_6.

In a query rule, there can be a considerable number of class membership literals, *In.c (x)*. They are introduced by three sources: the assignment of variables to classes, the instantiation of attribute value classes, and the interpretation of superclasses as classes of the answer variable.

Suppose that p is the OID of an object P (p, c, m, d) in the OB, and consider

A.p (x, y) $\wedge$ In.c (x)

then A.p (x, y) may be derived from two sources:

(1) axiom A_4, or

(2) deductive rule $\forall$ x/c', y/d' $\psi \Rightarrow$ A (x, m, y),

where A (x, m, y) is rewritten to A.p (x, y) - see Section 6.2.2.1.2.

In case (1), there must be an object P (o, x, I, y) with In (o, p). The attribute typing axiom A_6 then establishes the truth of In (x, c).

In case (2), c must be the lowest superclass of c' that has an attribute with label m. Thus In.c' (x) is true. As a consequence of axiom A_5, In.c (x) holds.

In either case, In.c (x) is already guaranteed by A.p (x, y), P (p, c, m, d), and the axioms of the OB. A similar argument holds for In.d(x) in the conjunction A.p(x, y) $\wedge$ In.d(y). Thus any conjunction:

A.p (x, y) $\wedge$ In.c (x), or
A.p (x, y) $\wedge$ In.d (y),

can be replaced by

A.p (x, y)

provided that p is the OID of an object P (p, c, m, d).

The above provides significant savings because an instantiation such as In.c(x) or In.d(y) will no longer be performed in the case of A.p (x, y) $\wedge$ In.c (x) or A.p (x, y) $\wedge$ In.d (y). [JS94] states that the efficiency increase via this elimination is a factor somewhere in the range of two to five. Also, since the axioms of the data model hold in any object database, this optimization can be applied to virtually any query, deductive rule, and integrity constraint inserted into the system.

6.2.2.3.2 Deductive Rule Optimization

One of the design objectives of O-Telos is to stay with the Datalog (with negation) frame work. This approach allows O-Telos to use virtually all existing optimization strategies developed for Datalog. In particular, O-Telos chooses to implement the Magic-Sets Rewrite algorithm to optimize the O-Telos queries (deductive rules). Magic-Sets was introduced in Section 6.1.2.3 and will not be presented again here.

6.2.2.3.3 Algebra Optimization

The object algebra, COBRA, was presented in German. There has been no English translation of this work and therefore the details of the optimization are not presented here. The current ConceptBase implementation of the optimizer is a direct implementation of the base literals *P*, *In*, *Isa*, and *A*. The implementation does not include algebra optimization. Algebraic expressions are directly evaluated via the Semi-Naive evaluation method [JS94].

6.2.2.3.4 Semi-Naive Evaluation

The final step in O-Telos query processing is the evaluation of algebraic expressions. The algorithm chosen by O-Telos is Semi-Naive evaluation. Semi-Naive evaluation is presented in Section 6.1.2.2.

7. Query Optimization for the Cyrano Prototype

In general, two approaches can be taken for Cyrano query optimization. One is to view Cyrano as an object-oriented model and take the object-oriented approach exemplified by SOSYS [SO90]. The other is to view Cyrano as a deductive object-oriented model and use the deductive object-oriented approach as exemplified by O-Telos [JS94]. The following subsections compare the two approaches from the point of view of query language characteristics as well as query optimization. The purpose of the comparison is to determine which query language is closer to that of Cyrano, and to find a query processing approach with query optimization strategies which can be adapted to the Cyrano prototype.

7.1 Query Language Comparisons

Table 2 shows the query language characteristics of SOSYS, O-Telos, and Cyrano. From the table, it can be seen that the SOSYS query language is a formal language and supports object-oriented features. The O-Telos query language is a formal language which supports object-oriented features as well as deductive features. Cyrano is a formal language which supports object-oriented and deductive features, even though it does not explicitly view queries as rules. Therefore, the O-Telos language is closer than that of SOSYS to the Cyrano query language. The following subsections make detailed comparisons between the query languages of SOSYS and O-Telos, SOSYS and Cyrano, and O-Telos and Cyrano.

Table 2 Comparison of Query Language Characteristics

Language Characteristics	SOSYS	O-Telos	Cyrano
Formal (F) Ad Hoc (A)	F	F	F
Behavior-Based (B) Structure-Based (S)	B	B	B
Object-Preserving (P) Object-Creating (G)	P	G	G
Query as Class (C)	C	C	C
Query as Deductive Rules (D)		D	
Support Recursive (R)		R	R
Support Quantifier (Q)		Q	Q

7.1.1 SOSYS versus O-Telos

There are three main differences between SOSYS and O-Telos:

- O-Telos supports recursive queries and SOSYS does not,
- O-Telos supports quantifiers and SOSYS does not, and
- O-Telos supports object-creation and SOSYS does not.

SOSYS follows an object-oriented approach based on relational query languages such as relational calculus and relational algebra, with object-oriented extensions. This kind of language usually does not support quantifiers and recursive queries, and is therefore less powerful than languages with the deductive object-oriented approach [Dzik96]. Straube does point out that, via some extension to the SOSYS languages, SOSYS could also support recursive and quantified queries.

Although SOSYS does not support object creation, some object-oriented models (e.g., [AA94]) do. Thus object-preserving is not necessarily a typical characteristic of the object-oriented approach, and SOSYS could well be extended to support object creation.

7.1.2 SOSYS versus Cyrano

The main differences between SOSYS and Cyrano are:

- Cyrano supports recursive queries and SOSYS does not,
- Cyrano supports quantifiers and SOSYS does not,
- Cyrano supports object creation and SOSYS does not, and
- Cyrano defines neither calculus nor algebra but SOSYS defines both.

Cyrano takes the deductive object-oriented approach. Therefore, it supports typical deductive language features such as quantifiers and recursive queries. In that respect, Cyrano is more powerful in expression than SOSYS. Moreover, Cyrano supports object creation. However, Cyrano does not provide a calculus or algebra, as SOSYS does.

7.1.3 O-Telos versus Cyrano

The main differences between O-Telos and Cyrano are:

- O-Telos views queries as classes as well as deductive rules. Cyrano views queries as classes but does not explicitly view queries as rules.
- O-Telos implements OID but Cyrano does not.
- O-Telos disallows the combination of objects from different classes, while Cyrano allows them, and
- O-Telos defines an object algebra but Cyrano does not.

Again, as in the case of comparing SOSYS and Cyrano, Cyrano is also less formal than O-Telos because Cyrano provides neither calculus nor algebra definitions. However, Cyrano does support features of deductive models such as quantifiers, recursive queries, and object creation. These features are similar to those of O-Telos. Both O-Telos and Cyrano support object creation, however, their definitions of object creation are different.

O-Telos supports object creation based on the concept of traditional deductive data models. Object creation is the process of deducing new objects based on existing objects and deductive rules. However, O-Telos does not allow the answer of a query to be a mixture of objects from different classes. For example, a query which requests a combination of objects from both class *Hammer* and class *Nail* is not allowed in O-Telos.

Object creation in Cyrano takes a different approach. Cyrano not only supports object creation based on the concept of traditional deductive models, but it also performs object creation in a new way, as defined in Cyrano research documents [DE94, Dzik96]. In Cyrano, the creation of a new object composed of objects from multiple classes is allowed. Therefore, queries that ask for combined objects of *Hammer-Nail* are permitted. In that respect, Cyrano provides a more powerful concept of object creation, and one which is different from traditional deductive models.

This more powerful support for object creation, however, does create problems in query optimization, because the typing of the new object becomes a difficult issue, i.e., proper typing is not guaranteed. Without the proper definition of typing, the application of subsequent methods to the new object becomes an issue. For example, after the creation of a *Hammer-Nail* object, applying the method of *wooden-handle* to the new object becomes improper, assuming that *wooden-handle* is a method of class *Hammer*.

To resolve the above problem, a sophisticated type checking system such as the one proposed by SOSYS is needed. Since such systems are still very much in the theoretical stage, this research will limit this kind of object creation to the final answer to queries. In other words, a query which derives a mixed-class object is allowed only if the base classes are not mixed classes. This avoids the issue of dealing with method application to the intermediate mixed-class objects.

7.2 Query Optimization Comparisons

The comparison of query optimization techniques will be based on two criteria. The first criterion considers the approaches a system takes to meet the challenges to object-oriented query optimization, as stated in Section 5.1. The second is whether the strategies used by a system for query optimization can be relatively easily adapted by Cyrano.

7.2.1 Meeting the Object-Oriented Query Optimization Challenges

Recall from Section 5.1 that there are mainly six typical areas which are difficult in object-oriented query optimization. These difficulties are introduced by the features of the object-oriented model. Table 3 lists the areas and shows whether a given system supports the given feature. The table indicates that Cyrano does not support any of these features. That is because the current Cyrano does not perform query optimization. The following subsections compare the approaches taken by SOSYS and O-Telos in supporting these features.

Table 3 Comparison of Query Optimization Techniques

	SOSYS	O-Telos	Cyrano
Method Optimization	Y	Y	N
Type Checking	Y	Y	N
Type Specific Optimization	Y	Y	N
Subtype Optimization	Y	Y	N
Common subexpression Optimization	Y	Y	N
OID Optimization	Y	Y	N

7.2.1.1 SOSYS Approaches

In SOSYS, the unit of a query is an atom. The expression of method such as *<object_x>.method* is considered to be an atom to the object calculus. Therefore, in the calculus optimization processing, methods are not optimized. But in the algebra optimization, methods are broken down into primitive operations, and the optimization of methods is therefore possible.

Type checking is performed in an innovative way in the SOSYS. In fact, one of the main contributions of SOSYS research is to guarantee the type consistency while not over restricting method application to the objects. An entire functional

area in the query processing methodology is devoted to type checking in order to ensure that the subsequent object algebra transformation will be performed on the type-consistent object algebraic expressions.

In SOSYS, type and subtype are expressed in a uniform way. Therefore optimization of subtype expressions includes no special treatment. Type and subtype optimizations are performed during the algebra optimization step.

Common subexpressions are eliminated during the calculus optimization (normalization). An OID represents an object in SOSYS, and queries are expressed in terms of OIDs. Therefore the query optimization process has no need to give a specific treatment for OIDs.

7.2.1.2 O-Telos Approaches

The beauty of O-Telos is its simplicity. Because of its total decomposition of the database into binary relations, because of its staying within the first-order logic, and especially due to its equivalence in expressive power to that of Datalog, O-Telos is able to perform object-oriented query optimization without any special treatment of methods, complex structures, and OIDs.

In O-Telos, queries are viewed as deductive rules. All predicates in the rules are binary relations which represent methods and complex structures. Thus, the optimization of the rules automatically takes care of the optimization of methods and complex structures, i.e., no specific mechanism is needed to optimize methods and complex structures. Common subexpressions appear in the form of predicates in the rules. Any duplicates of the predicates (common subexpressions) will be eliminated during the semantic / structural optimization of the rules.

OIDs are viewed as constants of predicates during the normalization process of structural optimization. Since OIDs are viewed as constants of the predicates, no special treatment is required during query optimization.

Type checking is relatively restricted in O-Telos. In O-Telos, an answer to a query can only take the type of the intersection of the superclasses of the query class. This is guaranteed by the axioms of the O-Telos language. As a result, all variables of the query are range restricted and the safety of a query is guaranteed. Also, query optimization will not encounter the improper application of methods due to query rewriting.

7.2.2 Comparisons of Query Optimization Strategies

Query optimization strategies taken by SOSYS and O-Telos were detailed in Section 5.3.3.3 and Section 6.2.2.3, respectively. The following briefly recaps both strategies.

SOSYS basically follows the traditional relational optimization strategies. It first normalizes a query expressed in the object calculus. The normalization is mainly for safety checking and for providing a common ground for translation to object algebra. The normalized calculus expression is then translated into object algebra. Type consistency checking is then applied to the object algebraic expression. The type-consistent algebraic expression is further optimized by rewriting, using algebraic equivalence rules. Costs are then associated to different expressions. The best (least cost) path associated with a particular expression is then chosen for evaluation.

O-Telos basically follows the traditional Datalog optimization strategies. O-Telos first simplifies a query expressed in the deductive rules. The simplified expression is then optimized via the Magic-Sets Rewrite method. The optimized expressions are then translated into the object algebra. The algebra is further optimized via algebraic rewriting. The optimized algebraic expressions are then evaluated via the Semi-Naive evaluation method. The strategies of SOSYS and O-Telos are shown in Figure 27.

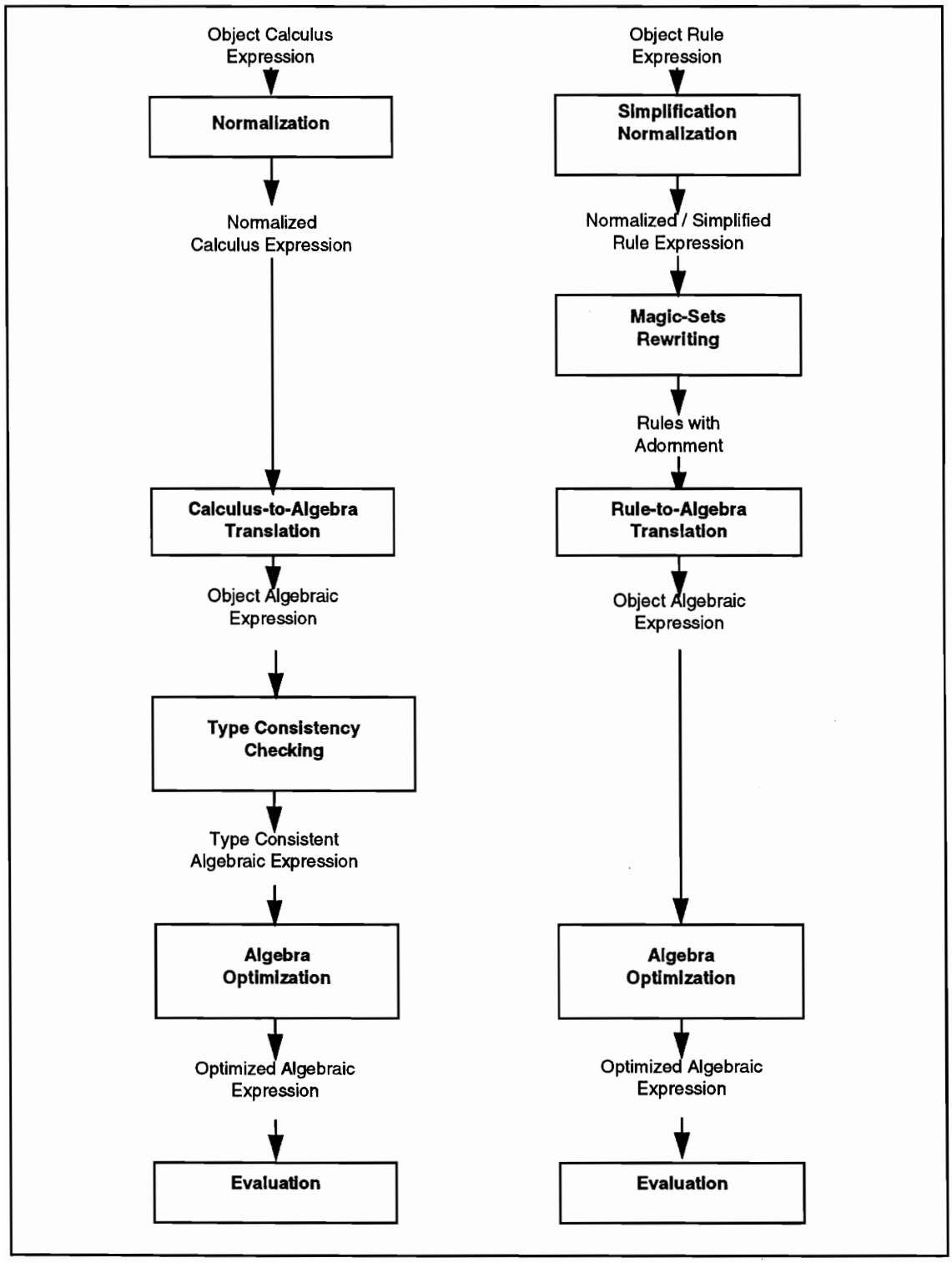

Figure 27 Query Processing Methodologies - SOSYS vs. O-Telos

As can be seen from the figure, the two strategies are very similar in terms of the overall approach, even though the algorithms used in each system are very different. Both systems take a query expressed in a declarative language (calculus/ rule), normalize the expression, translate the normalized expression into algebraic expressions, optimize the algebraic expression, and then evaluate the query.

In SOSYS, Magic-Sets rewriting is not needed because SOSYS query processing performs early binding of constants, which is an often used technique in relational query processing methodologies. Magic-sets rewriting is needed for O-Telos because O-Telos uses the *bottom-up* evaluation method. *Bottom-up* evaluation involves no early binding, and therefore side-pass binding information can be used in order to improve the efficiency of query evaluation.

7.3 Selection of Query Optimization Strategy for Cyrano

The analysis in Section 7.1 and Section 7.2 shows that the O-Telos language is closer than SOSYS to the Cyrano language because both O-Telos and Cyrano take the deductive object-oriented approach. Both Cyrano and O-Telos view queries as classes, however, O-Telos also views queries as rules. In addition, both O-Telos and Cyrano take the bottom-up evaluation approach. This makes it easier for Cyrano to adapt the O-Telos optimization strategies.

In general, the differences between Cyrano and O-Telos are smaller than the differences between Cyrano and SOSYS. Currently, Cyrano views queries as classes but not as deductive rules. To make Cyrano also view queries as rules, Cyrano query classes can be translated into deductive rules. Section 7.7.3 pro-

vides an algorithm for the translation. The main purpose for the translation is to enable Cyrano to use an array of working optimization algorithms developed for deductive data models.

Cyrano research supports OID. However, OID is not implemented in the Cyrano prototype. Since the Cyrano prototype guarantees the uniqueness of object labels, labels will be used as object identifiers.

It is relatively difficult to make Cyrano closer to SOSYS so as to utilize the query processing methodology from SOSYS, mainly because SOSYS does not support features such as object creation, quantifiers, and recursive queries which are commonly used by deductive models. In order to support these features, new operators need to be added, new constraints need to be devised, and new transformation/rewriting rules need to be introduced.

From the query optimization point of view, both SOSYS and O-Telos made an effort to meet the challenges introduced by object-oriented features. As can be seen in Section 7.2.1, O-Telos takes a simpler approach. The reason that O-Telos is able to do so is due to its language design, which had simplicity as one of its main goals.

Another comparison is that O-Telos has been implemented in a prototype called ConceptBase which has been widely used for research and development. The implementation further proves the feasibility of the O-Telos concept. In contrast, based on the best knowledge of this research, there has been no implementation of SOSYS.

Based on the above analysis, Cyrano query processing in general and query optimization in particular will take an approach similar to that of O-Telos, i.e., to use deductive object-oriented query processing with deductive query optimization and bottom-up evaluation strategies.

7.4 Proposed Query Processing Methodology for Cyrano

The proposed query processing methodology for the Cyrano prototype is shown in Figure 28. There are two main differences between this methodology and the one used by the original Cyrano prototype. The first is the addition of the Magic-Sets rewriting, and the second is the replacement of the Naive evaluation with Semi-Naive evaluation.

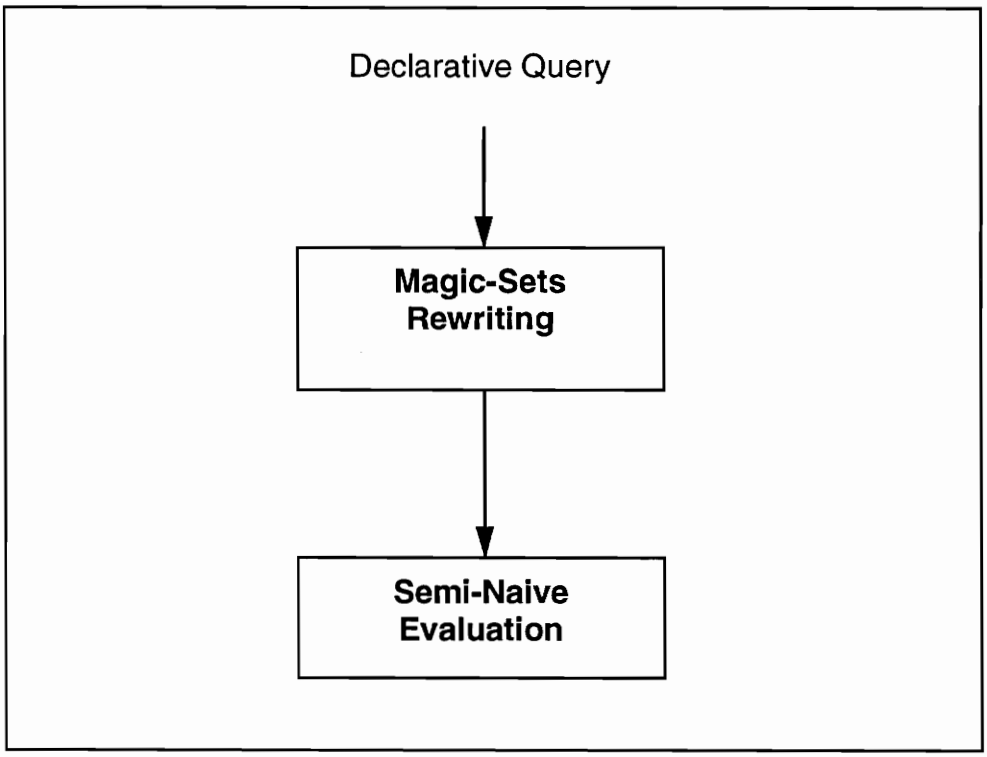

Figure 28 New Cyrano Query Processing Methodology

Figure 29 breaks down the query processing in order to show detail. There are still two main stages in Cyrano query processing. The first stage is the *compilation* stage, depicted in the upper box; the second stage is the *execution* stage, appearing in the lower box.

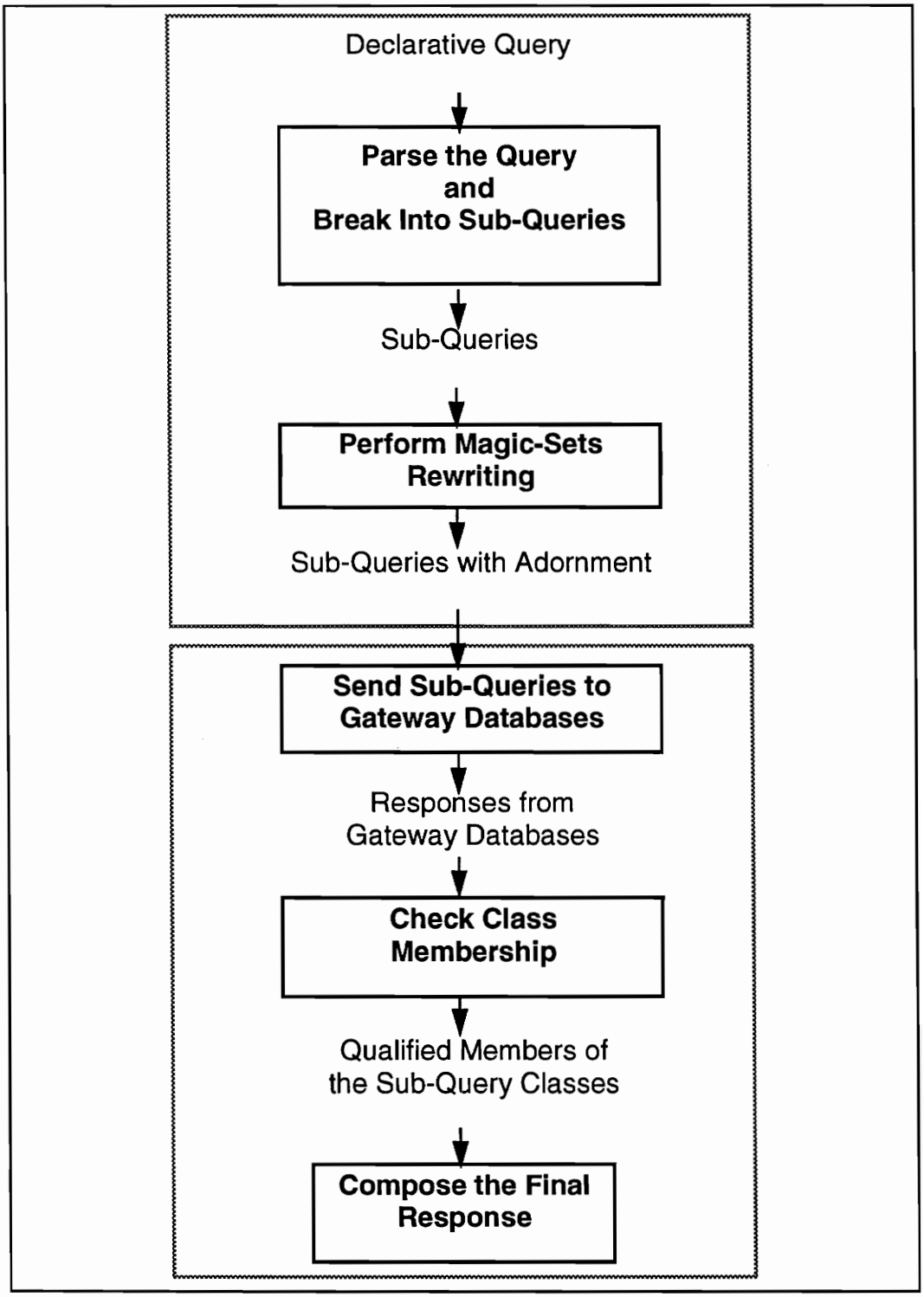

Figure 29 New Cyrano Query Processing Steps

7.4.1 Compilation Stage

In the compilation stage, parsing and translating of the query into the Cyrano internal class structures are still the same as they are in the original Cyrano. However, Magic-Sets rewriting will be added to reduce the size of the relevant objects and thus make execution more efficient. Note that Magic-Sets rewriting is performed in the compilation stage and therefore requires no additional time during execution.

7.4.2 Execution Stage

The new execution stage is similar to that of the original Cyrano. The main difference between the two is that the Naive evaluation method is replaced by the Semi-Naive evaluation method. Figure 30 shows the improved Cyrano query evaluation algorithm, which implements the lower box (execution stage) of Figure 29.

```
Done = FALSE;
DO UNTIL Done
    Done = TRUE;
    DO for all classes in the Result-List
        IF this is a gateway class
            IF this is the first time in the loop for this class
                Evaluate the class
                Add the result to the Result-List
                Done = FALSE
            ENDIF
        ELSE
            DO for every derivation of the class for which the computation
                    of the derivation has not been completed
                Find all combinations of the base objects of the class
(##)            which contain at least one new object
                    generated from the last DO UNTIL Done loop
                DO for all such combinations
                    IF a combination satisfies the guard of the derivation
(#)                     IF the new object is not already on the Result-List
                            Generate a derived object from the combination
                            Add it to the Result-List
                            Done = FALSE
                        ENDIF
                    ENDIF
                ENDDO
            ENDDO
        ENDIF
    ENDDO
ENDDO
```

Figure 30 New Cyrano Query Evaluation Algorithm

There are two main differences between the new algorithm (Figure 30) and the original Cyrano algorithm (Figure 12). The first difference (marked by #) is that the new algorithm guarantees termination. The algorithm will add a new object to the Result-List only if the object is not already on the list. The algorithm continues if there is at least one new object added to the Result-List during an iteration. The algorithm stops when an iteration did not add any new object to the Result-List.

The second difference (marked by ##) is that the Semi-Naive evaluation is used to replace the original Naive evaluation. Computation at the Nth iteration will only be performed on the object set which contains at least one new object generated from the (N-1)th iteration. This algorithm thus effectively eliminates the recomputation efforts made by the Naive algorithm. Application of this algorithm yields up to several hundred percent performance improvement over the original Cyrano query processing.

Additional performance improvement is achieved by computing only those derivations which have not completed computation. This modification to the original algorithm eliminates the wasted effort of repeating the computation for computation-completed derivations.

7.5 State of Implementation

The Semi-Naive Evaluator was chosen to be implemented in order to serve as the proof of the concept: query processing in Cyrano FDBS can be optimized, and the optimization algorithms developed for traditional data models can be used to improve the performance of the Cyrano FDBS.

One reason for choosing to implement the Semi-Naive Evaluator is the need to improve the performance of recursive query processing. The original Cyrano prototype can process a non-recursive query involving ten derivations with ten gateway classes and a total of 5000 objects in 40 seconds. However, for a recursive query involving only two derivations with one gateway class and a total of 20 objects, it would take the original Cyrano about 1800 seconds to process. The need for improving the recursive query processing performance is evident, and the Semi-Naive algorithm fits the purpose well, because it was designed for the sole purpose of improving the performance of recursive query processing.

7.6 Results of the Implementation

The Semi-Naive Evaluator was implemented for the Cyrano global database query process. The implementation yielded up to several hundred percent performance improvement for processing recursive queries in comparison to the original Cyrano. In order to compare the performance of the new Cyrano with the original Cyrano, the termination condition was added to the original Cyrano.

7.6.1 Testing Environment

Testing was performed on a 486-66 MHz PC with 8 megabytes of main memory. The Borland C++ Profiler was used to measure the performance.

7.6.2 Testing Procedures

For each test case, the following steps were performed:

- Install the database for the test case.

- Reset the PC.
- Load the original Cyrano prototype.
- Perform the test and collect data.
- Reset the PC.
- Load the new Cyrano prototype.
- Perform the test and collect data.

The reason for resetting the PC before each test was to prevent inaccurate data collection introduced by factors such as memory fragmentation caused by a previous test.

Multiple runs were performed for each test case. Data from each run was accumulated by the Profiler. The final result was the average of the output of the multiple runs. Number of runs were determined by the time interval of a single run. The following lists the number of runs in relation to the time intervals (t). The time unit is in seconds.

- $t \leq 12$: 20 runs
- $12 < t \leq 60$: 10 runs
- $60 < t \leq 120$: 7 runs
- $120 < t \leq 600$: 5 runs
- $600 < t$: 3 runs

The choice of the number of runs was based on (1) Borland C++ Profiler allowed a maximum of 20 cumulative runs, and (2) the precision of the Profiler was one ten-thousandth of a second. Since the minimum time per run obtained from the test data was about three seconds, 20 runs would give an accumulated time of 60 seconds, which should be sufficient in accuracy with the one ten-thousandth of a second precision.

7.6.3 Testing Results

7.6.3.1 Non-Recursive Queries

Since the Semi-Naive algorithm was designed for improving the performance of processing recursive queries, the implementation of this algorithm has no effect on the processing of non-recursive queries. The slight gain in performance improvement for non-recursive queries is due to the elimination of the redundant computation for the computation-completed derivations.

As shown in Figure 31, the gateway database for the test has 10 classes. Each class has 500 corresponding objects. Thus the database has a total of 5000 objects.

```
class student is gateway
  with
       STRING ss_num;
       STRING name;
       STRING major;
       STRING gpa;
       STRING year;
       STRING sex;
end class.

class employee is gateway
  with
       STRING ss_num;
       STRING name;
       STRING dept;
       STRING salary;
       STRING sex;
end class.

class course is gateway
  with
       STRING title;
       STRING student_name;
       STRING grade;
       STRING required_for;
end class.

class library_book is gateway
  with
       STRING title;
       STRING call_num;
       STRING subject;
       STRING borrowed_by;
end class.

class department is gateway
  with
       STRING name;
       STRING professor;
end class.
```

```
class faculty is gateway
    with
        STRING ss_num;
        STRING name;
        STRING dept;
        STRING status;
end class.

class club is gateway
    with
        STRING name;
        STRING num_members;
end class.

class classroom is gateway
    with
        STRING building_num;
        STRING room_num;
        STRING days;
end class.

class pc is gateway
    with
        STRING serial_num;
        STRING brand;
        STRING cpu;
end class.

class piano is gateway
    with
        STRING serial_num;
        STRING brand;
        STRING type;
end class.
```

Figure 31 Database for Non-Recursive Query Testing

The testing queries in the Cyrano global class expression are given by Figure 32. The first query is a derived class with five derivations. Each derivation is derived from a gateway database with a *selection* operation. The second query is similar to the first, except that ten derivations are used instead of five.

```
class Q1 is derived with
   student s: (s.major = "MATH")
   with
        student s1 is s;
   end;

   employee e: (e.dept = "CS")
   with
        employee e1 is e;
   end;

   course co: (co.required_for = "PHYS")
   with
        course c1 is co;
   end;

   department d: (d.name = "PE")
   with
        department d1 is d;
   end;

   library_book lb: (lb.subject = "ENGL")
   with
        library_book lb1 is lb;
   end;
end class.
```

(a)

```
class Q2 is derived with
     student s: (s.major = "MATH")
     with
          student s1 is s;
     end;

     employee e: (e.dept = "CS")
     with
          employee e1 is e;
     end;

     course co: (co.required_for = "PHYS")
     with
          course c1 is co;
     end;

     department d: (d.name = "PE")
     with
          department d1 is d;
     end;

     library_book lb: (lb.subject = "ENGL")
     with
          library_book lb1 is lb;
     end;

     faculty f: (f.status = "FULL_TIME")
     with      faculty f1 is f;
     end;

     club cl: (cl.num_members = "55")
     with      club cl1 is cl;
     end;

     classroom cr: (cr.days = "MWF")
     with      classroom cr1 is cr;
     end;

     pc p: (p.cpu = "Pentium")
     with      pc p1 is p;
     end;

     piano pi: (pi.brand = "KIMBALL")
     with      piano pi1 is pi;
     end;
end class.
```

(b)

Figure 32 Query Classes for Non-Recursive Query Testing

Table 4 shows the performance comparison of the original Cyrano and the current Cyrano for non-recursive query processing. As can be seen from the table, even with 10 derivations, the performance improvement is only about five percent.

Table 4 Performance Comparison for Non-Recursive Queries

Number of Derivations	Execution Time Original Cyrano (seconds)	Execution Time New Cyrano (seconds)	Ratio Original : New	Improvement (%)
5	21.7850	20.7650	1.0491:1	4.912
10	39.2660	37.2050	1.0554:1	5.540

7.6.3.2 Recursive Queries

The performance improvement in the processing of recursive queries is significant due to the implementation of the Semi-Naive algorithm. Two sets of test were done in order to investigate the performance of the implementation. In the first set of the test, the database only contains two relevant objects, all other objects in the database are irrelevant to the query. During query processing, the second iteration of the algorithm generates a new object. The third iteration generates no new objects and the query processing stops.

In the second set of the test, the database contains more relevant objects to the query. During query processing, each iteration generates some new objects. The processing stops after all existing objects are exhausted.

Figure 33 (a) is the database for the first test set, and Figure 33 (b) is the database for the second test set. In the first set of test, up to 300 objects were used. In the second set of test, however, only 20 objects were used due to the very long execution time required for processing the query.

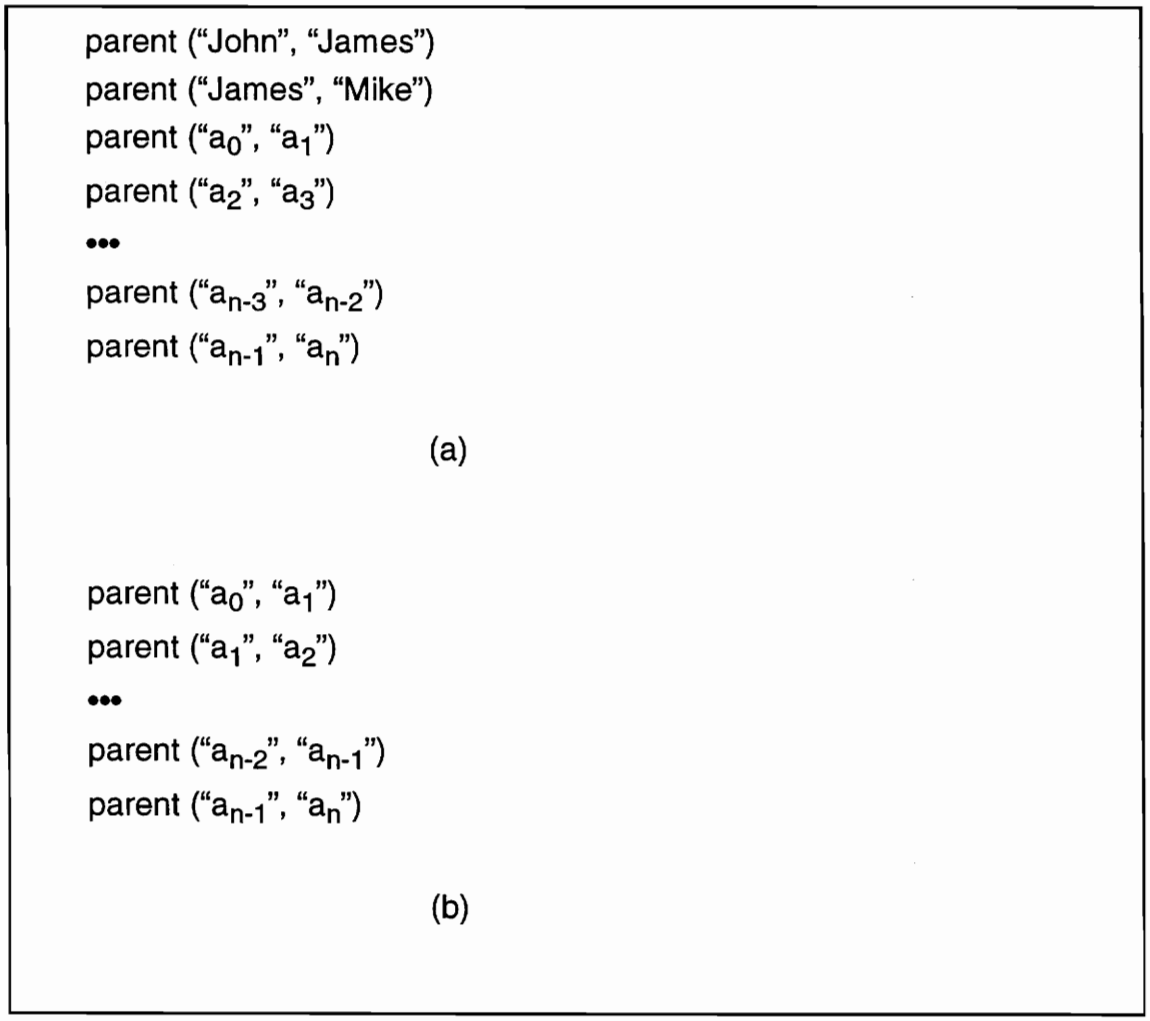

Figure 33 Database for Recursive Query Testing

Figure 34 shows two test queries. The first query, Q3, asks for all ancestors of John, and the second query, Q4, asks for all ancestors of a_0. The gateway class *parent* is defined in Figure 34 (c), and the *ancestor* rules expressed in a Cyrano class are defined in Figure 34 (d). Class *ancestor* is equivalent to the *ancestor* rules below:

R1: ancestor (X, Y):- parent (X, Y)
R2: ancestor (X, Y):- ancestor (X, Z), parent (Z, Y)

The reason for choosing the ancestor query as the testing query is that it has been used as the testing query for performance benchmarks of deductive databases (e.g., [BR89]).

```
class Q3 is derived with
    ancestor a1: (a1.descendent = "John")
    with
        STRING name is a1.ancest;
    end;
end class.
```

(a)

```
class Q4 is derived with
    ancestor a2: (a2.descendent = "a0")
    with
        STRING name is a2.ancest;
    end;
end class.
```

(b)

```
class parent is gateway
    with
        STRING young;
        STRING old;
end class.
```

(c)

```
class ancestor is derived with
    parent p: TRUE
    with
        STRING descendent is p.young;
        STRING ancest is p. old;
    end;
    ancestor a, parent p: (a.ancest = p.young)
    with
        STRING descendent is a.descendent;
        STRING ancest is p.old;
    end;
end class.
```

(d)

Figure 34 Query Classes for Recursive Query Testing

To demonstrate the differences in processing between the two queries, Figure 35 lists the objects generated from a few iterations of the processing of the two queries via Semi-Naive evaluation.

Q3:

Iteration 1

("John", "James"), ("James", "Mike"), ("a_0", "a_1"), ("a_2", "a_3")...

("a_{n-1}", "a_n")

Iteration 2

("John", "Mike")

Iteration 3

Null.

(a)

Q4:

Iteration 1

("a_0", "a_1"), ("a_1", "a_2"), ("a_2", "a_3"), ("a_3", "a_4"), ("a_4", "a_5")...

Iteration 2

("a_0", "a_2"), ("a_1", "a_3"), ("a_2", "a_4"), ("a_3", "a_5")...

Iteration 3

("a_0", a_3"), ("a_1", "a_4"), ("a_2", "a_5")...

•••

(b)

Figure 35 Iterations of Recursive Query Process Testing

Table 5 Performance Comparison for Recursive Query - Q3

Database Size (number of objects)	Execution Time Original Cyrano (seconds)	Execution Time New Cyrano (seconds)	Ratio Original : New	Improvement (%)
10	5.7771	3.7544	1.5388:1	53.88
20	10.8368	5.7216	1.8940:1	89.40
50	42.9506	16.5486	2.5954:1	159.54
100	154.8720	57.3230	2.7017:1	170.17
200	603.4400	207.7650	2.9044:1	190.44
300	1279.8000	440.3100	2.9066:1	190.66

Table 6 Performance Comparison for Recursive Query - Q4

Database Size (number of objects)	Execution Time Original Cyrano (seconds)	Execution Time New Cyrano (seconds)	Ratio Original : New	Improvement (%)
10	79.8893	15.9394	5.0121:1	401.21
20	1773.6000	162.6200	10.9064:1	990.64

Table 5 shows the performance comparison between the original Cyrano and the improved Cyrano for processing Q3. Table 6 shows the performance comparison between the original Cyrano and the improved Cyrano for processing Q4. The power of a better algorithm is well documented by the two tables. In Q3 processing, fewer iterations were executed, i.e., after iteration 2, new object ("John", "Mike") was generated, iteration 3 generated no new object, and the execution stopped. In this case, the performance improvement ranges from an adequate 50 percent to a significant 190 percent.

In Q4 processing, much more redundant work was done by the original Cyrano since each iteration in this case generated some new objects until the entire database was exhausted. The improvement produced by the new Cyrano was a stunning 990 percent with a database of only 20 objects.

The testing results also indicate that the Semi-Naive algorithm was properly implemented in the new Cyrano. The implementation achieved significant performance improvement, which was in agreement with [BR89].

Figure 36 depicts the performance improvement based on the data from Table 5. The figure indicates that as the number of objects increases, percentage improvement increases as well. However, there was almost no improvement when the number of objects increased from 200 to 300. This is because in the case of processing Q3, as the number of objects in the databases increases, so does the number of irrelevant objects. Recall that the Semi-Naive algorithm eliminates redundant computation, but it does not eliminate irrelevant objects. When the number of irrelevant objects increases, other algorithms such as Magic-Sets rewriting are needed in order to achieve further performance improvement.

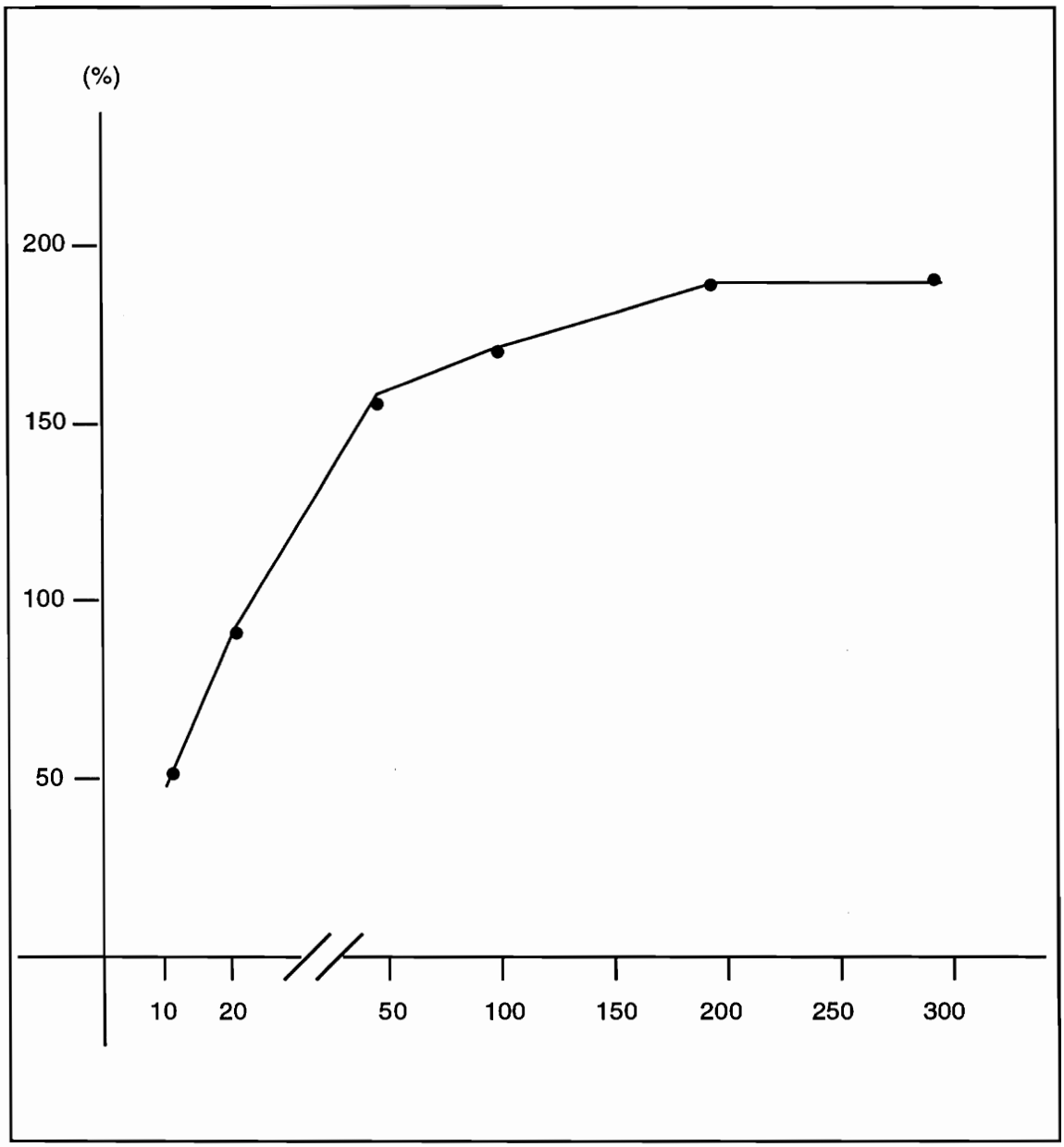

Figure 36 Performance Improvement Curve - Q3

7.6.3.3 Magic-Sets Rewriting

The Magic-Sets Rewrite algorithm was not implemented by this research. However, to prove that the Magic-Sets Rewrite algorithm can further improve the performance, the *ancestor* class was rewritten based on the Magic-Sets Rewrite algorithm. Query evaluation is still via the Semi-Naive algorithm. The databases are the same databases as shown in Figure 33. Queries Q3 and Q4 are again used to query the database. Q3 and Q4 are now derived from *magic_ancestor* class rather then the original *ancestor* class, where *magic_ancestor* is rewritten from *ancestor* via the Magic-Sets Rewrite method.

Figure 37 shows the steps of the Magic-Sets rewriting and the output from the rewriting. The steps are according to the algorithm given in Section 6.1.2.3. The query *Q (y)* is Q3 in the figure. Q4 can also be used by replacing Q^f *(y):- ancestor*^{bf} *("John", y)* with Q^f*(y):- ancestor*^{bf} *("*a_0*", y)*. The corresponding Cyrano classes Q3, Q4, and *magic_ancestor* are shown in Figure 38. For convenience in reference, class *parent* is also listed in Figure 38.

Reachable Adorned System:

Q^f (y):- ancestor^{bf} ("John", y)

ancestor^{bf} (x, y):- parent (x, y)

ancestor^{bf} (x, y):- ancestor^{bf} (x, z), parent (z, y)

Output from Magic-Sets Rewriting:

magic_R0_ancestor^{bf} ("John").

magic_R2_ancestor^{bf} (x):- magic_ancestor^{bf} (x).

Q^f (y):- magic_R0_ancestor^{bf} ("John"), ancestor^{bf} ("John", y)

ancestor^{bf} (x, y):- parent (x, y).

ancestor^{bf} (x, y):- magic_R2_ancestor^{bf} (x), ancestor^{bf} (x, z), parent (z, y).

magic_ancestor^{bf} ("John"):- magic_R0_ancestor^{bf} ("John").

magic_ancestor^{bf} (x):- magic_R2_ancestor^{bf} (x).

Renaming magic_Rn_ancestor^{bf} (x) to magic (x):

magic ("John").

Q^f (y):- ancestor^{bf} ("John", y).

ancestor^{bf} (x, y):- parent (x, y).

ancestor^{bf} (x, y):- magic (x), ancestor^{bf} (x, z), parent (z, y).

Final Output:

Q (y):- ancestor ("John", y).

ancestor (x, y):- parent (x, y).

ancestor ("John", y):- ancestor ("John", z), parent (z, y).

Figure 37 Magic-Sets Rewriting for Ancestor Query

```
class Q3 is derived with
    magic_ancestor ma1: (ma1.descendent = "John")
    with
        STRING name is ma1.ancest;
    end;
end class.
```

(a)

```
class Q4 is derived with
    magic_ancestor ma1: (ma1.descendent = "a0")
    with
        STRING name is ma1.ancest;
    end;
end class.
```

(b)

```
class magic_ancestor is derived with
    parent p: TRUE
    with
        STRING descendent is p.young;
        STRING ancest is p. old;
    end;
    magic_ancestor ma, parent p: ((ma.descendent = "John") and (ma.ancest = p.young))
    with
        STRING descendent is ma.descendent;
        STRING ancest is p.old;
    end;
end class.
```

(c)

```
class parent is gateway
    with
        STRING young;
        STRING old;
end class.
```

(d)

Figure 38 Query Classes After Magic-Sets Rewriting

Table 7 shows a performance comparison between Magic-Sets Rewriting with Semi-Naive evaluation and pure Semi-Naive evaluation for Q3 processing. The table indicates that when the database contains 300 objects, the performance improvement is 2695 percent. The performance improvement is not only impressive but also makes the execution time more tolerable for the user. Although Semi-Naive evaluation gained a 190 percent performance improvement over the Naive method when there were 300 objects, it still took 440 seconds for the Semi-Naive method to give an answer. When Semi-Naive evaluation is used with Magic-Sets rewriting, the execution time is reduced to 16 seconds, which is certainly more acceptable to the users.

Table 7 Magic-Sets + Semi-Naive vs. Semi-Naive - Q3

Database Size (number of objects)	Execution Time Semi-Naive (seconds)	Execution Time Magic-Sets + Semi-Naive (seconds)	Ratio Semi-Naive : Magic-Sets + Semi-Naive	Improvement (%)
10	3.7544	3.0334	1.2377:1	23.77
20	5.7216	3.5547	1.6096:1	60.96
50	16.5486	4.9815	3.3220:1	232.20
100	57.3230	7.3766	7.7709:1	677.09
200	207.7650	11.7633	17.6621:1	1666.21
300	440.3100	15.7527	27.9514:1	2695.14

Table 8 does a performance comparison between Magic-Sets rewriting with the Semi-Naive evaluation and Naive evaluation for Q3 processing. An 8000 percent performance improvement is achieved for a database containing 300 objects.

Table 8 Magic-Sets + Semi-Naive vs. Naive - Q3

Database Size (number of objects)	Execution Time Original Cyrano (seconds)	Execution Time Magic-Sets + Semi-Naive (seconds)	Ratio Original Cyrano : Magic-Sets + Semi-Naive	Improvement (%)
10	5.7771	3.0334	1.9045:1	90.45
20	10.8368	3.5547	3.0486:1	204.86
50	42.9506	4.9815	8.6220:1	762.20
100	154.8720	7.3766	20.9950:1	1999.50
200	603.4400	11.7633	51.2985:1	5029.85
300	1279.8000	15.7527	81.2432:1	8024.32

Table 9 shows a performance comparison between Magic-Sets rewriting with Semi-Naive evaluation and Semi-Naive evaluation for Q4 processing. As expected, Magic-Sets rewriting does not give much performance improvement, because most of the objects in this test are relevant to the query.

Table 9 Magic-Sets + Semi-Naive vs. Semi-Naive - Q4

Database Size (number of objects)	Execution Time Semi-Naive (seconds)	Execution Time Magic-Sets + Semi-Naive (seconds)	Ratio Semi-Naive : Magic-Sets + Semi-Naive	Improvement (%)
10	15.9394	15.3630	1.0375:1	3.75
20	162.6200	135.990	1.1958:1	19.58

The above experiments show that Magic-Sets rewriting together with Semi-Naive evaluation could achieve the optimization purpose via both the reduction of the relevant facts and the elimination of redundant computation. Magic-Sets rewriting can achieve significant performance improvement, especially when the size of relevant objects is small in comparison to the size of the irrelevant objects. Since common queries only access a small portion of the existing objects, Magic-Sets rewriting will in general improve the efficiency of query processing.

7.7 Further Improvement

One obvious next step for further performance improvement of the Cyrano query processing is the implementation of the Magic-Sets Rewrite algorithm. According to [BR89], Magic-Sets rewriting with Semi-Naive evaluation will give a performance improvement over the Naive evaluation by orders of magnitude for the *ancestor* query over a database of 100,000 objects. Experiments conducted as part of this research show that significant performance improvement is achieved by using the Magic-Sets Rewrite method, especially where the size of relevant objects is smaller than the size of irrelevant objects.

Since enterprise databases are usually large databases, and the size of the relevant objects is usually much smaller then the size of the irrelevant objects, implementing the Magic-Sets rewriting method is advisable. In addition, this optimization is performed during query compilation time, therefore, the optimization will only need to be performed once, and the resulting optimized query can be executed as many times as needed. With orders of magnitude savings in execution time, the performance improvement can be very significant.

Further improvement may be achieved by implementing a query processing methodology which is closer to the one used by O-Telos. Figure 39 depicts the proposed methodology. The reason for taking the extra step of translating query classes to rules is to enable further exploration of the query structures, so as to facilitate further optimization via query structure manipulation. Deductive query optimization algorithms, including the Magic-Sets Rewrite method, can be more conveniently implemented when queries are expressed in terms of deductive rules, because they were developed for rule based systems.

In order to perform the class-to-rule transformation, a new BNF and a class-to-rule transformation algorithm are needed. The following subsections introduce functional areas of the proposal, suggest a new BNF, present a possible transformation algorithm, and discuss structural optimization.

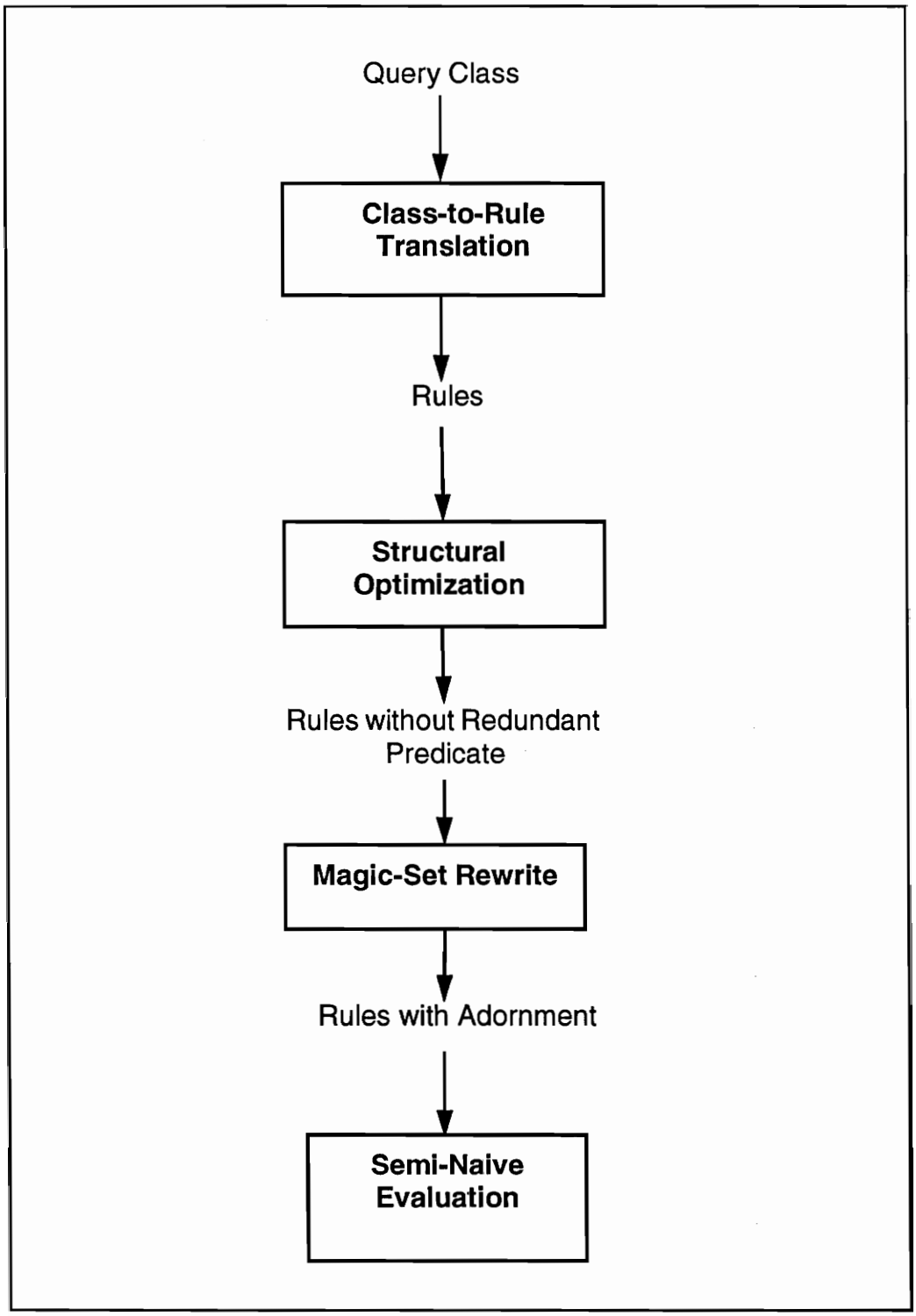

Figure 39 Proposed Methodology for Further Improvement

7.7.1 Functional Areas

The following functional areas (components) are needed in order to implement fully the proposed Cyrano query processing methodology depicted in Figure 39.

- A BNF for Cyrano query classes. This BNF will be based on O-Telos BNF and the current BNF for Cyrano.
- Class-to-rule translator. The translator converts an input query class to equivalent deductive rules by using the class-to-rule transformation algorithm presented in Section 7.7.3.
- Structural optimizer: This optimizer takes a deductive rule as input, finds duplicated predicates and typed attributes, and then eliminates the duplicated predicates and simplifies the typed attributes. The output is a deductive rule without duplicated predicates and with simplified attributes.
- Magic-Sets rewriting optimizer: The optimizer takes the output from the structural optimizer and performs Magic-Sets rewriting processing. The output is a set of rules with Magic-Sets adornments.
- Semi-Naive evaluator: This evaluator takes adorned rules as input, and performs an evaluation based on the Semi-Naive algorithm.

7.7.2 New BNF for the Cyrano Query Language

The new BNF is a first step in the development of the Class-to-Rule translator. The BNF is based on the O-Telos BNF as well as the current Cyrano BNF. In the new Cyrano BNF, rules will be written in rule expressions rather than in class expressions. This will make it easier for a user to write a query. For example, it is much easier to write the *ancestor* rules in the rule format than in the class format.

The O-Telos explicit definitions of *attribute* and *constraint* will replace the implicit Cyrano definitions for *attributes* and *guards*. The Cyrano derived class definition is used in addition to the O-Telos superclass definition, and the derivation is made explicit by using the keyword *isDerived* as opposed to *isA*. The derived class definition is needed because Cyrano is an FDBS, and Cyrano uses derivation to resolve the heterogeneity problem, which may not be an issue in O-Telos. The new Cyrano BNF is given in Appendix F.

7.7.3 Query Class to Rule Transformation

The first step in the query processing is to translate a query class into deductive rules. Two types of attributes need to be defined before the introduction of the translation algorithm.

The type-1 attribute is an instantiation of an attribute of a superclass (or base class) of the query class. A type-1 attribute is written as *a:S*, where *a* is the attribute name and *S* is one of the superclasses (or base classes) of the query class.

The type-2 attribute is an attribute of the query class. It is written as *b:T*, where *b* is the attribute name and *T* is the type of *b*. The following is the class-to-rule transformation algorithm.

1. For each superclass C_i of the query class, add $In.C_i$ (o) to the body of the rule, where *o* is the answer object.

2. For each base class D_i of the query class, add $In.D_i$ (o) to the body of the rule, where *o* is the answer object.

3. For a type-1 attribute $a_i:S_i$, add a conjunction $In.S_i$ $(u_i) \wedge A.a_i$ (o, u_i) to the body of the rule, where *o* is the answer object, a_i is a type-1 attribute, S_i is the name of the superclass (or base class) of a_i, and u_i is a new variable.

4. For a type-2 attribute $b_i:T_i$, add $In.T_i$ (v_i) to the body of the rule, where v_i is a new variable, and T_i is the type of b_i.

5. Substitute the newly introduced variables u_i, v_i, and *o* for their symbolic counterparts a_i, b_i, and *this*, thus transforming the logical formula describing the rules and constraints.

6. Add universal quantifiers for the answer object and all new variables.

7. Use the answer object and new variables as the arguments of the query predicate.

8. Link all atoms with conjunctions.

The following gives two examples of the translation. The first is the translation of a generic query class into rules, and the second is a specific query class to rule translation. In the translations, the steps shown correspond to steps of the algorithm. If a step is not applicable to a specific query, it is so stated.

Class Q5 isA C_1,..., C_k isDerived D_1,..., D_k with

attribute

a_1: S_1;... a_m: S_m;

b_1: T_1;... b_n: T_n;

constraint

c: $<formula> $

end.

Translation

1. $In.C_1(o)$,..., $In.C_j(o)$

2. $In.D_1(o)$,..., $In.D_k(o)$

3. $In.S_1(u_1) \wedge A.a_1(o, u_1)$,..., $In.S_m(u_m) \wedge A.a_m(o, u_m)$

4. $In.T_1(v_1)$,... $In.T_n(v_n)$

5. ψ (not applicable here).

6. $\forall o, u_1,..., u_m, v_1,..., v_n$

7. $Q(o, u_1,..., u_m, v_1,..., v_n)$

8. $\forall o, u_1,..., u_m, v_1,..., v_n$

$In.C_1(o) \wedge ... \wedge In.C_j(o) \wedge In.D_1(o) \wedge ... \wedge In.D_k(o) \wedge$

$In.S_1(u_1) \wedge A.a_1(o, u_1) \wedge ... \wedge In.S_m(u_m) \wedge A.a_m(o, u_m) \wedge$

$In.T_1(v_1) \wedge ... \wedge In.T_n(v_n) \wedge \psi \Rightarrow Q5(o, u_1,..., u_m, v_1,..., v_n)$

Figure 40 Class-to-Rule Translation Example 1

Q6 is a query to find every student who takes a class which is not required by his major. Class *Student* is assumed to have attributes *takes* and *majors*, and class *Major* is assumed to have attribute *requires*. Figure 41 defines the query class, Q6, and shows the translation steps.

Class Q6 isA Student with

attribute

wrongCourse: Course;

constraint

wc: $A (this, takes, wrongCourse) and

not exits m/Major and A (this, majors, m) and

A (m, requires, wrongCourse) $

end.

Translation

1. In.Student(o)
2. Not applicable due to no base class
3. Not applicable due to no type-1attribute
4. In.Course (v_1)
5. A.takes (o, v_1) $\wedge \neg \exists$m In.Major(m) $\wedge$ A.majors (o, m) $\wedge$ A.requires (m, v_1)
6. $\forall$ o, v_1
7. Q6 (o, v_1)
8. $\forall$ o, v_1 In.Student (o) $\wedge$ In.Course (v_1) $\wedge$
   A.takes (o, v_1) $\wedge \neg \exists$m In.Major(m) $\wedge$ A.majors (o, m) $\wedge$ A.requires (m, v_1)
   $\Rightarrow$ Q6 (o, v_1)

Figure 41 Class-to-Rule Translation Example 2

7.7.4 Structural Optimization

Once the class-to-rule translation is implemented, structural optimization can be implemented. Structural optimization can give performance improvement by a factor of 2 to 5. Also, structural optimization can be performed during the compilation time so that no additional time is needed during query execution. Once the query is compiled, it can be executed many times without having to re-compile it for each execution. Because this optimization can reduce the execution time by a factor of 2 to 5, the extra compilation time for the optimization is negligible.

8. Conclusions

8.1 Summary

The purpose of this research is to improve the performance of query processing in the Cyrano prototype. The Cyrano prototype is an FDBS with a deductive object-oriented meta model. The mechanism for achieving the performance improvement is query optimization.

The thesis introduces the basics of query, query language, query processing, and query optimization. It details the original Cyrano query processing methodology and points out the deficiencies in the methodology. The thesis provides surveys of the existing query processing methodologies and query optimization techniques for FDBSs, deductive databases, and object-oriented databases. It also presents challenges for object-oriented query optimization. The thesis compares the methodologies and techniques, and selects those that can be adapted to the Cyrano prototype. The thesis proposes a new query processing methodology for the Cyrano prototype. It reports the status of the implementation of this methodology, and presents the results of its application. Finally, the thesis gives recommendations for possible further improvement of the Cyrano query processing methodology, and concludes that the performance of the Cyrano query processing can be significantly improved by query optimization.

8.2 Contributions

The thesis proposed a new query processing methodology with query optimization for the Cyrano FDBS prototype. In the new query processing methodology, Semi-Naive algorithm replaced the Naive algorithm of the original Cyrano, and Magic-Sets Rewrite algorithm was added. Both Semi-Naive and Magic-Sets Rewrite algorithms are query optimization techniques. Semi-Naive algorithm eliminates the redundant computation done by the Naive algorithm, and Magic-Sets Rewrite algorithm reduces the number of irrelevant objects in the computation. The result was a new Cyrano query processor which significantly outperformed the original Cyrano query processor. The performance improvement was up to several hundred percent. The *ancestor* query was used as the testing query because this query had been used as the testing query for performance benchmarks of deductive databases. Testing results showed that the new Cyrano outperformed the original Cyrano in each test case. In particular, when the input data caused the original Cyrano to perform the most amount of redundant computation, the new Cyrano yielded a 400% performance improvement with a database of only 10 objects.

Query optimization for an FDBS with a deductive object-oriented meta model is a new area in FDBS query optimization, because most existing query optimization techniques are focused on the FDBSs with relational meta models. Having shown that the performance of the Cyrano query processing can be improved by query optimization will likely lead to renewed interest in query optimization in FDBSs with non-relational meta models. In addition, the new Cyrano query processing methodology is a general deductive object-oriented query processing methodology, and it can well be applied to other FDBSs with deductive object-oriented meta models.

The thesis shows that, in general, the query optimization techniques developed for traditional deductive and object-oriented models can be applied to the Cyrano prototype, a prototype FDBS with a deductive object-oriented meta model. This provides possibilities for other FDBSs with deductive object-oriented meta models to use an array of well established query optimization techniques.

8.3 Future Research Directions

This section identifies areas of future research, which are related to but not resolved by this research.

The functional areas presented in Section 7.7 need to be further investigated and refined before implementation. For example, the thesis presents an example that uses a typing axiom to perform structural optimization. This could serve as a starting point for structural optimization. In order to take full advantage of the structural optimization, however, query structures need to be further investigated, and more axioms need to be developed.

Static type checking is an important and difficult issue in object-oriented query optimization.The powerful Cyrano object creation concept will be fully realized only if the type checking issue is resolved. As discussed in the thesis, Cyrano takes a new and more powerful approach to object creation. This approach, however, creates a new problem in query optimization, because the typing of the new object is difficult. This problem makes query transformation very difficult, if not impossible. In order to solve the problem, an advanced type checking system needs to be designed and developed.

Although their use is complicated, constraints are very useful in query optimization. As discussed in the thesis, proper applications of constraints may produce answers to a query without any database computation. Query optimization using constraints deserves further research effort because it can bring potential performance improvement.

Distributed query optimization issues also deserve investigation. One of the Cyrano research goals is to have a distributed Cyrano FDBS. To fully realize this goal, distributed query optimization issues will have to be addressed and resolved.

Bibliography

[AA94] R. Alhajj and M. Arkun. A Formal Object-Oriented Query Model and an Algebra. In Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Springer-Verlag, 1994, pp. 101-116.

[ACM90] Special Issue on Multidatabases. ACM Computing Surveys, 22:3, 1990.

[Back78] J. Backus. Can Programming be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs. Communications of the ACM 21:8, 1978, pp. 613-641.

[BB93] R. Bal and H. Balsters. A Deductive and Typed Object-Oriented Language. In Proceedings of the third International Conference on Deductive and Object-Oriented Databases, Springer-Verlag, 1993, pp. 340-358.

[BBV90] H. Balsters, R. de By, and C. de Vreeze. The TM Manual version 1.2. Manuscript, University of Twente, Enschede, 1990.

[BD90] V. Benzaken and C. Delobel. Enhancing Performance in a Persistent Object Store: Clustering Strategies in O2. In Implementing Persistent Object Bases: Principles and Practice. Proceedings of Fourth International Workshop on Persistent Object Systems. San Mateo, CA: Morgan Kaufmann, 1990.

[Beer94] C. Beeri. Query Languages for Models with Object-Oriented Features. In Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Springer-Verlag, 1994, pp. 47-71.

[Bert91] E. Bertino. Method Precomputation in Object-Oriented Databases. SIGOIS Bulletin, 12:2, 1991, pp. 199-212.

[Bert94] E. Bertino. A Survey of Indexing Techniques for Object-Oriented Database Management Systems. In Query Processing for Advanced Database Systems. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

[BK89] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. IEEE Transactions on Knowledge and Data Engineering, 1:2, 1989, pp 196-214.

[BK93] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. Theoretical Computer Science 116, 1993, pp. 59-94.

[BR89] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In Readings in Artificial Intelligence and Databases. San Mateo, CA: Morgan Kaufmann Publishers, 1989.

[Catt95] R. Cattell. The Object Database Standard: ODMG-93, release 1.2. San Mateo, CA: Morgan Kaufmann, 1995.

[CD92] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In Proceedings of ACM SIGMOD International Conferences on Management of Data, 1992, pp. 225-236.

[CD94] S. Cluet and C. Delobel. Towards a Unification of Rewrite-Based Optimization Techniques for Object-Oriented Queries. In Query Processing for Advanced Database Systems. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. Logic Programming and Databases. Germany: Springer-Verlag, 1990.

[Codd70] E. Codd. A Relational Model of Data for Large Shared Data Banks. CACM 13, No.6, June 1970.

[Codd72] E. Codd. Relational Completeness of Data Base Sub-languages. In Database Systems: Courant Computer Science Symposia Series 6, Englewood Cliffs, NJ: Perentice Hall, 1972.

[CWN94] S. Chakravarthy, W. Whang, and S. Navathe. A Logic-Based Approach to Query Processing in Federated Databases. Information Sciences, Vol. 79, 1994, pp. 1-28.

[DGKM91] S. Daniels., G. Graaff, T. Keller, D. Maier, D. Schmidt, and B. Vance. Query Optimization in Revelation, an Overview. IEEE Database Engineering, 14:2. 1991, pp. 58-62.

[Date91] C. Date. An Introduction to Database Systems, Vol. 1, Reading, MA: Addison-Wesley Publishing Co., 1991.

[DE94] J. Dzikiewicz and C. Egyhazy. Cyrano: A Meta Model for Federated Database Systems. In Proceedings of ISMM International Conference, June 1994, pp. 52-55.

[DKS92] W. Du, R. Krishnamurthy, and M. Shan. Query Optimization in Heterogeneous DBMS. In Proceedings of the 18th VLDB Conference, August 1992, pp. 277-291.

[Dzik96] J. Dzikiewicz. Meta Models for Federated Database Systems. Ph.D. dissertation, Virginia Tech, 1996.

[FG91] B. Finance and G. Gardarin. A Rule-Based query Rewriter in an Extensible DBMS. In Proceedings of 7th International Conference on Data Engineering, 1991, pp. 248-256.

[Gard94] G. Gardarin. Object-Oriented Rule Languages and Optimization Techniques. In Advances in Object-Oriented Database Systems. Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Springer-Verlag, 1994, pp. 225-250.

[GCDM94] G. Graefe, R. Cole, D. Davison, W. McKenna, R. Wolniewicz. Extensible Query Optimization and Parallel Execution in Volcano. In Query Processing for Advanced Database Systems. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

[GW89] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In Proceedings of ACM SIGMOD International Conference on Management of Data, 1989, pp. 358-366.

[HB96] A. Hurson and M. Bright. Object-Oriented Multidatabase Systems. In Object-Oriented Multidatabase Systems - A Solution for Advanced Applications. Englewood Cliffs, NJ: Prentice Hall, 1996.

[IEEE91] Special Issue on Multidatabases. IEEE Computer, Dec. 1991.

[JGJS95] M. Jarke, R. Gallersdorfer, M. Jeusfeld, and M. Staudt. ConceptBase - A Deductive Object Base for Meta Data Manager. Journal of Intelligent Information Systems, 4:2, 1995, pp. 167-192.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. ACM Computing Survey, 16:2, June 1984, pp. 112-152.

[JS94] M. Jeusfeld and M. Staudt. Query Optimization in Deductive Object Bases. In Query Processing for Advanced Database Systems. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

[JWKL90] B. Jenq, D. Woelk, W. Kim, and W. Lee. Query Processing in Distributed ORION. In Advances in Database Technology. Springer-Verlag, 1990, pp. 167-187.

[Kim91] W. Kim. Introduction to Object-Oriented Databases. Cambridge, Mass: MIT Press, 1991.

[KM90] A. Kemper and G. Moerkotte. Access Support in Object Bases. In Proceedings of ACM SIGMOD International Conference on Management of Data, May 1990, pp. 364-374.

[LCL95] C. Lee, C. Chen, and H. Lu. An Aspect of Query Optimization in Multidatabase Systems. ACM SIGMOD, 14:3, 1995, pp. 28-33.

[LVZ92] R. Lanzelotte, P. Valduriez, and M. Zait. Optimization of Object-Oriented Recursive Queries Using Cost-Controlled Strategies. In Proceedings of ACM SIGMOD International Conference on Management of Data, 1992, pp. 256-265.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge About Information Systems. ACM Transactions on Information Systems, 8:4, 1990, pp. 325-362.

[MS93] J. Melton and A. Simon. Understanding the New SQL: A complete Guide. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[MZD94] C. Mitchell, S. Zdonik, and U. Dayal. Optimization of Object-Oriented Query Languages: Problems and Approaches. In Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Springer-Verlag, 1994, pp. 119-146.

[MSTB94] D. Maier, S. Daniels, T. Keller, B. Vance, G. Graefe, W. McKenna. Challenges for Query Processing in Object-Oriented Databases. In Query Processing for Advanced Database Systems. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

[Nils80] N. Nilsson. Principles of Artificial Intelligence. Palo Alto, CA: Tioga Publishing Co., 1980.

[Ozsu91] M. Ozsu. Query Processing Issues in Object-Oriented Database Systems - Preliminary Ideas. In Proceedings of the 1991 Symposium on Applied Computing, April 1991, pp. 312-324.

[PZ91] M. Palmer and S. Zdonik. Fido: A Cache that Learns to Fetch. In Proceedings of 17th International Conference on Very Large Databases, 1991. pp 255-264.

[SJJN93] M. Staudt, M. Jarke, M. Jeusfeld, and H. Nissen. Query Classes. Deductive and Object-Oriented Databases. In Proceedings of the third DOOD International Conference, Springer-Verlag,1993, pp. 283-295.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Survey, 22:3, 1990, pp. 183-236.

[SO90] D. Straube and M. Ozsu. Queries and Query Processing in Object-Oriented Database Systems. ACM Transactions on Information Systems, 8:4, 1990, pp. 387-430.

[Stra91] D. Straube. Queries and Query Processing in Object-Oriented Database Systems. Ph.D. dissertation, University of Alberta, 1991.

[Theo92] H. Thoennissen. Design and Implementation of an Object Algebra for a Deductive Object Base System (in German), Diploma Thesis, RWTH Aachen, Germany, 1992.

[Ullm88] J. Ullman. Principles of Database and Knowledge-Base Systems. Rockville, MD: Computer Science Press, 1988.

[Zdon94] S. Zdonik. What Makes Object-Oriented Database Management Systems Different? In Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Springer-Verlag, 1994, pp. 1-26.

Appendix A. Current BNF for Cyrano Query Language

```
<CLASS>
    ::= "CLASS" <NAME> "IS"
            (<DERIVED_DEFINITION> | <GATEWAY_DEFINITION>)
        "END" "CLASS" "."

<DERIVED_DEFINITION>
    ::= "DERIVED" "WITH" {<DERIVATION> ";"}

<DERIVATION>
    ::= [<Variable> {"," <VARIABLE>}] ":" <VALUE>
        "WITH" {<DERIVED_METHOD> ";"}
        "END"

<VARIABLE>
    ::= <CLASS_NAME><VAR_NAME>

<CLASS_NAME>
    ::= <NAME>

<VAR_NAME>
    ::= <NAME>

<DERIVED_METHOD>
    ::=<CLASS_NAME> <NAME>
        ["("[<VARIABLE> {","<VARIABLE>}] ")"] "IS" <VALUE>

<VALUE>
    ::= <BASE_VALUE> {<COMPLEX_VALUE>}
        | <COMPLEX_VALUE>
```

```
<BASE_VALUE>
    ::= "(" <VALUE> ")"
    |   <CONSTANT_VALUE>
    |   <VARIABLE_VALUE>
    |   <QUANTIFIED_VALUE>
    |   <COND_VALUE>

<CONSTANT_VALUE>
    ::= <Quoted_String>
    |   <Integer>
    |   <True>
    |   <False>

<COMPLEX_VALUE>
    ::=  "." <NAME> ["(" [<VALUE> {"," <VALUE>}] ")"]
    |   <OPERATOR> <VALUE>
    |   "NOT" <VALUE>

<QUANTIFIED_VALUE>
    ::= ("EXISTS" | "FORALL")
        "(" <VARIABLE> ")" <VALUE>

<COND_VALUE>
    ::= "IF" "(" { "(" <VALUE> "THEN" <VALUE> ")"} ")"

<OPERATOR>
    ::= "+" | "-" | "*" | "/"
    |   "=" | "<" | ">" | "<>" | "<=" | ">="
    |   "AND" | "OR"

<GATEWAY_DEFINITION>
    ::= "GATEWAY" <GATEWAY_TYPE> "WITH" {<METHOD> ";"}
```

<GATEWAY_TYPE>
    ::= ":" "PARADOX" | ":" "DATALOG"

<METHOD>
    ::= <CLASS_NAME><NAME> ["(" [<INTEGER>] ")"]

Appendix B. SOSYS - Calculus-to-Algebra Transformation Example

This appendix gives an example of transformation from object calculus expression to object algebra expression. The example is taken from [SO90].

Query

Find all nodes belonging to the structural part of a document authored by a person who is retired.

Object Calculus Expression

{o | ∃p (Doc (p)
∧ ∃q ("65" = q ∧ "True" = <p, q>.author.age.greater
∧ ∃r (StructLink (r) ∧ p == <r>.part_of
∧ (o == <r>.from ∨ o == <r>.to))))}

Transformation

- Number the atoms:

  a1 Doc (p)
  a2 "65" = q
  a3 "True" = <p, q>.author.age.greater
  a4 StructLink (r)
  a5 p == <r>.part_of
  a6 o == <r>.from
  a7 o == <r>.to

- Delete the constant defining atoms and replace all occurrences of the variables with the corresponding constants:

  a1 Doc (p)
  a3 "True" = <p, "65">.author.age.greater
  a4 StructLink (r)
  a5 p == <r>.part_of
  a6 o == <r>.from

a7 o == <r>.to

- Link atoms with conjunction or disjunction based on the original query:
  D1 ≡ a1 ∧ a3 ∧ a4 ∧ a5 ∧ a6
  D2 ≡ a1 ∧ a3 ∧ a4 ∧ a5 ∧ a7

The following uses D1 to illustrate the rest of the steps of the transformation.

- Re-write the atoms in the shorthand form:
  a1 (p | _) ∧ a3 (_ | p) ∧ a4 (r | _) ∧ a5 (p | r) ∧ a6 (o | r)

- Re-write range atoms from a (v | _) to a (_ | v)
  a1 (p | _) ∧ a3 (_ | p) ∧ a4 (r | _) ∧ a5 (_ | p, r) ∧ a6 (o | r)

- Recursively call function Place () and yield the following nested expression of atoms:
  a6 (o | r) ∧ (a5 (_ | p, r) ∧ (a3 (_ | p) ∧ (a1 (p | _))$_p$)$_p$ ∧ (a4 (r | _))$_r$)$_r$

- Map the above expression to the object algebra counterparts:
  (StructLink σ_{a5} <$_($ Doc σ_{a3} < >)>) γ^t_{a6} < >

Appendix C. SOSYS - Algebraic Rewriting Rules

NOTATIONS

ref (F, v_1,..., v_n) is true when v_1,..., v_n are the only variables referenced in the predicate F.

gen (F, v) is true when the predicate F contains a generating atom for the variable v.

res (F, v) is true when the predicate F restricts values of v.

$E_1 \Leftrightarrow E_2$ denotes that E_1 is equivalent to E_2.

$E_1 \Leftrightarrow^c E_2$ denotes that E_1 is equivalent to E_2 only when condition c is true.

IDENTITY REWRITE RULES

Set Identities

$P \cup (Q \cup R) \Leftrightarrow (P \cup Q) \cup R$

$P \cup (Q \cap R) \Leftrightarrow (P \cup Q) \cap (P \cup R)$

$P - (Q \cup R) \Leftrightarrow (P - Q) \cap (P - R)$

$P - (Q \cap R) \Leftrightarrow (P - Q) \cup (P - R)$

$P \cap (Q \cup R) \Leftrightarrow (P \cap Q) \cup (P \cap R)$

$P \cap (Q \cap R) \Leftrightarrow (P \cap Q) \cap R$

$P - (Q - R) \Leftrightarrow (P - Q) \cup (P \cap Q \cap R)$

$(P - Q) - R \Leftrightarrow (P - Q) \cap (P - R)$

$(P \cup Q) - R \Leftrightarrow (P - R) \cup (Q - R)$

$(P \cap Q) - R \Leftrightarrow (P - R) \cap (Q - R)$

Commutativity of Select

$(P\ \sigma_{F1} < Qset >)\ \sigma_{F2} < Rset > \Leftrightarrow (P\ \sigma_{F2} <Rset>)\ \sigma_{F1} <Qset>$

Commutativity of Difference with Respect to Select

$(P - Q)\ \sigma_F < Rset> \Leftrightarrow (P\ \sigma_F <Rset>) - Q$

Distributivity of Union with Respect to Select - A

$(P \cup Q)\ \sigma_F\ \langle Rset \rangle \Leftrightarrow (P\ \sigma_F\ \langle Rset \rangle) \cup (Q\ \sigma_F\ \langle Rset \rangle)$

Distributivity of Union with Respect to Select - B

$P\ \sigma_F\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \Leftrightarrow$

$(P\ \sigma_F\ \langle Q_1 \ldots Q_x \ldots Q_k \rangle) \cup (P\ \sigma_F\ \langle Q_1 \ldots Q_x \ldots Q_k \rangle)$

Distributivity of Intersect with Respect to Select

$(P \cap Q)\ \sigma_F\ \langle Rset \rangle \Leftrightarrow (P\ \sigma_F\ \langle Rset \rangle) \cap (Q\ \sigma_F\ \langle Rset \rangle)$

Distributivity of Union with Respect to Generate - A

$(P \cup Q)\ \gamma^{\dagger}_F\ \langle Rset \rangle \Leftrightarrow (P\ \gamma^{\dagger}_F\ \langle Rset \rangle) \cup (Q\ \gamma^{\dagger}_F\ \langle Rset \rangle)$

Distributivity of Union with Respect to Generate - B

$P\ \gamma^{\dagger}_F\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \Leftrightarrow$

$(P\ \gamma^{\dagger}_F\ \langle Q_1 \ldots Q_x \ldots Q_k \rangle) \cup (P\ \gamma^{\dagger}_F\ \langle Q_1 \ldots Q_y \ldots Q_k \rangle)$

Distributivity of Union with Respect to Map - A

$(P \cup Q) \rightarrow_{mlist} \langle Rset \rangle \Leftrightarrow (P \rightarrow_{mlist} \langle Rset \rangle) \cup (Q \rightarrow_{mlist} \langle Rset \rangle)$

Distributivity of Union with Respect to Map - B

$P \rightarrow_{mlist} \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \Leftrightarrow$

$(P \rightarrow_{mlist} \langle Q_1 \ldots Q_x \ldots Q_k \rangle) \cup (P \rightarrow_{mlist} \langle Q_1 \ldots Q_y \ldots Q_k \rangle)$

SELECT REWRITE RULES

Fractorization of Cascaded Selects

$(P\sigma_{F1}\ \langle Qset \rangle)\ \sigma_{F2}\ \langle Rset \rangle \Leftrightarrow (P\ \sigma_{F1}\ \langle Qset \rangle) \cap (P\ \sigma_{F2}\ \langle Rset \rangle)$

Conjunctive Select Predicate -A

$P\ \sigma_{(F1 \wedge F2)}$ <Qset, Rset> $\Leftrightarrow^{c}$ (P σ_{F1} <Qset>) $\cap$ (P σ_{F2} <Rset>)

where:

c: ref(F1, (p, $q_1...q_k$)) $\wedge$ res(F1, p) $\wedge$ ref (F2, (p, $r_1...r_l$)) $\wedge$ res (F2, p)

Conjunctive Select Predicate - B

$P\ \sigma_{(F1 \wedge F2)}$ <Qset, R, Sset> $\Leftrightarrow^{c1}$ P σ_{F1} <Qset, (R σ_{F2} <Sset>)>

$\Leftrightarrow^{c2}$ P σ_{F1} <Qset, (R γ^{t}_{F2} <Sset>)>

where:

c1: ref(F1, (p, $q_1...q_k$, r)) $\wedge$ ref (F2, (r, $s_1...s_m$)) $\wedge$ res (F2, r)

c2: ref(F1, (p, $q_1...q_k$, t)) $\wedge$ ref (F2, (t, r, $s_1...s_m$)) $\wedge$ gen (F2, t)

Disjunctive Select Predicates

$P\ \sigma_{(F1\ v\ F2)}$ <Qset, Rset> $\Leftrightarrow^{c}$ (P σ_{F1} <Qset>) $\cup$ (P σ_{F2} <Rset>)

where:

c: ref(F1, (p, $q_1...q_k$)) $\wedge$ res(F1, p) $\wedge$ ref (F2, (p, $r_1...r_l$)) $\wedge$ res (F2, p)

Factoring Generate from a Conjunctive Select

$P\ \sigma_{(F1 \wedge F2)}$ <Qset, R, Sset> $\Leftrightarrow^{c}$ (P σ_{F1} <Qset>) $\cap$ (R γ^{p}_{F2} <Sset>)

where:

c: ref(F1, (p, $q_1...q_k$)) $\wedge$ res(F1, p) $\wedge$ ref (F2, (p, r, $s_1...s_m$)) $\wedge$ gen (F2, p)

GENERATE REWRITE RULES

Conjunctive Generate Predicates

$P\ \gamma^{t}_{(F1 \wedge F2)}$ <Qset, Rset> $\Leftrightarrow^{c1}$ (P σ_{F1} <Qset>) γ^{t}_{F2} <Qset>

$\Leftrightarrow^{c2}$ (P γ^{u}_{F1} <Qset>) γ^{t}_{F2} <Rset>

$\Leftrightarrow^{c3}$ (P γ^{t}_{F1} <Qset>) σ_{F2} <Rset>

where:

c1: ref(F1, (p, q_1...q_k)) $\wedge$ res(F1, p) $\wedge$ ref (F2, (p, r_1...r_l, t)) $\wedge$ gen (F2, t)

c2: ref(F1, (p, q_1...q_k, u)) $\wedge$ gen(F1, u) $\wedge$ ref (F2, (r_1...r_l, t, u)) $\wedge$ gen (F2, t)

c3: ref(F1, (p, q_1...q_k, t)) $\wedge$ gen(F1, t) $\wedge$ ref (F2, (r_1...r_l, t)) $\wedge$ res (F2, t)

Factoring Generate from a Conjunctive Generate

P $\gamma^t_{(F1 \wedge F2)}$ <Qset, R, Sset> $\Leftrightarrow^c$ ((P γ^r_{F1} <Qset>) $\cap$ R) γ^t_{F2} <Sset>

where:

c: ref(F1, (p, r, q_1...q_k)) $\wedge$ gen(F1, r) $\wedge$ ref (F2, (r, t, s_1...s_m)) $\wedge$ gen (F2, t)

SEMANTIC REWRITE RULES

In the following, c_i denotes a class, C_i denotes the extent of c_i, and C^*_i denotes the deep extent of c_i. The notation $c_1 \nabla c_2$ states that $\exists c_i \mid c_i \leq c_1 \wedge c_i \leq c_2$, i.e., C^*_1 and C^*_2 have elements in common, where $c_1 \leq c_2$ denotes that all members of c_1 are members of c_2. The notation Δ denotes $\neg \nabla$, and the notation c_1 $\neg\leq c_2$ means that c_1 and c_2 have no members in common.

$C_1 \cap C_2 = 0$ when $c_1 \neq c_2$

$C_1 - C_2 = C_1$ when $c_1 \neq c_2$

$C_1 \cap C^*_2 = C_1$ when $c_1 \leq c_2$

$C_1 \cap C^*_2 = 0$ when $c_1 \Delta c_2$

$C_1 \cup C^*_2 = C^*_2$ when $c_1 \leq c_2$

$C_1 - C^*_2 = C1$ when $c_1 \neq c_2 \wedge c_1 \neg\leq c_2$

$C^*_1 - C_2 = C^*_1$ when $c_1 \Delta c_2$

$C^*_1 \cap C^*_2 = 0$ when $c_1 \Delta c_2$

$C^*_1 \cup C^*_2 = C^*_2$ when $c_1 \leq c_2$

$C_1^* - C^*_2 = C^*_1$ when $c_1 \Delta c_2$

$C^*_1 \cap C^*_2 = \cup C_i$, $c_i \leq c_1 \wedge c_i \leq c_2$ when $c_1 \nabla c_2$

Appendix D. Datalog Basics

This appendix gives a brief introduction to the Datalog query language. The materials used are based on [CGT90].

VARIABLE

VARIABLE consists of all finite alphanumeric character strings beginning with an upper-case letter.

CONSTANT

CONSTANT consists of all finite alphanumeric character strings which are either numeric or consist of one lower-case letter followed by zero or more lower-case letters or digits.

PREDICATE

PREDICATE consists of all finite alphanumeric character strings beginning with a non-numeric lower-case character. Predicates and Constants have a non-empty intersection; however, in a Datalog program, it is always clear from the context whether a symbol stands for a predicate or a constant.

TERM

A term is either a constant or a variable. A term t is a *ground* iff it is a constant. The set Constants of all ground terms is also called the *Herbrand Universe*.

ATOM

An atom $p\ (t_1,\ldots,\ t_n)$ consists of an n-ary predicate symbol p and a list of arguments $(t_1,\ldots,\ t_n)$, such that each t_i is a term. A ground atom is an atom which contains only constants as arguments.

LITERAL

A literal is an atom $p\ (t_1,\ldots,\ t_n)$ or a negated atom $\neg\ p\ (t1,\ldots,\ tn)$. A ground literal is a ground atom or a negated ground atom. Literals which are atoms are also called positive literals, and literals consisting of negated atoms are called negative literals.

CLAUSE

A clause is a finite list of literals, or equivalently, an ordered set with the possibility of duplicates and not merely as a set. A ground clause is a clause which is a finite set of ground literals. A unit clause is a clause consisting of one single literal. Clauses containing only negative literals are called negative clauses, and clauses containing only positive literals are called positive clauses.

A Horn clause is a clause containing at most one positive literal. Datalog uses Horn clauses to express database facts, rules, and the query (goal).

- Facts: Facts are positive unit clauses. They express unconditional knowledge. For example, *parent (jim, john)* states the fact that *john* is a parent of *jim*.

- Rules: A rule is a clause with exactly one positive literal and with at least one negative literal. A rule represents conditional knowledge of the type - *if p is true then q is true.* For example, *{¬ ancestor (X, Z), ancestor (X, Y), ¬ parent (Z, Y)}*, which says if *Y* is a parent of *Z*, and *Z* is an ancestor of *X*, then *Y* is an ancestor of *X*. In Datalog syntax, this rule can be expressed as: *ancestor (X, Y):- ancestor (X, Z), parent (Z, Y).*

- Goal: A goal is a negative clause, i.e., a clause consisting only of negative literals. Unit goals are goals with only one literal. For example, *{¬ parent (jim, X)}*, which asks who is the parent of *jim*. In Datalog syntax, this is expressed as *"?- parent (jim, X)*", or in a query format such as "*query (X):- parent (jim, X).*"

  The names *Goal* or *Goal clause* come from resolution theorem proving. In resolution theorem proving, in order to prove that a certain formula holds, the negation of this formula is first assumed to be true. The proving procedure then shows that this assumption leads to some contradiction, thus proving that the original formula is true. For example, if we want to prove the formula $\exists X$ *(parent (jim, X))*, the negation of the formula is first taken: $\forall X$ (¬ *parent (jim, X)*. In Datalog syntax, this is expressed as *"query (X):- parent (jim, X).*"

Appendix E. Magic-Sets Rewriting Example

1. Original Program:

R0: q (y):- ancestor (a, y)
R1: ancestor (x, y): - parent (x, y)
R2: ancestor (x, y):- ancestor (x, z), parent (z, y)

2. Adorned Program:

R0: q^{f} (y):- $ancestor^{bf}$ (a, y)

For Adornment (b, f) of ancestor

R1: $ancestor^{bf}$ (x, y): - parent (x, y)

R2: $ancestor^{bf}$ (x, y):- $ancestor^{bf}$ (x, z), parent (z, y)

For Adornment (f, b) of ancestor

R1: $ancestor^{fb}$ (x, y): - parent (x, y)

R2: $ancestor^{fb}$ (x, y):- $ancestor^{fb}$ (x, z), parent (z, y)

3. Distinguished Arguments

From R0 ^{f}: *a* because *a* is a constant.

From R1 ^{bf}: *x* because *x* is bounded by *b* of the head of the rule.

: *y* because *parent* is an EDB predicate and *parent* has a distinguish argument *x*.

From R2 ^{bf}: *x* because *x* is bounded by *b* of the head of the rule.

4. Reachable Adorned System (P^{A})

R0: q^{f}(y):- $ancestor^{bf}$ (a, y)

R1: $ancestor^{bf}$ (x, y):- parent (x, y)

R2: $ancestor^{bf}$ (x, y):- $ancestor^{bf}$ (x, z), parent (z, y)

5. Generate Magic Sets Program (P^{magic})

$P^{magic} = P^{A}$

LOOP_1:

From R0: q^{f}(y):- ancestor ^{bf} (a, y)

IDB predicate = ancestor

a) NA

b) magic_R0_ancestor^{bf} (a, y)

c) magic_R0_ancestor^{bf} (a)

d) NA

e) NA

f) NA

g) R01: magic_R0_ancestor ^{bf} (a)

h) $P^{magic} = P^{magic}$ + R01

From R1: ancestor ^{bf} (x, y):- parent (x, y)

No IDB predicate.

From R2: ancestor ^{bf} (x, y):- ancestor ^{bf} (x, z), parent (z, y)

IDB predicate = ancestor

a) NA

b) magic_R2_ancestor ^{bf}(x, z), parent (x, y)

c) magic_R2_ancestor ^{bf}(x), parent (x, y) // z is not distinguished

d) magic_R2_ancestor ^{bf}(x) // parent () is not distinguished

e) magic_ancestor ^{bf} (x, y):- magic_R2_ancestor ^{bf}(x)

f) magic_ancestor ^{bf} (x):- magic_R2_ancestor ^{bf}(x) // y is not distinguished

g) R21: magic_R2_ancestor ^{bf}(x):- magic_ancestor ^{bf} (x)

h) $P^{magic} = P^{magic}$ + R21

P^{magic} after LOOP_1:

R0: q^{f} (y):- ancestor ^{bf} (a, y)

R1: ancestor ^{bf} (x, y): - parent (x, y)

R2: ancestor ^{bf} (x, y):- ancestor ^{bf} (x, z), parent (z, y)

R01: magic_R0_ancestor ^{bf} (a)

R21: magic_R2_ancestor ^{bf}(x):- magic_ancestor ^{bf} (x)

LOOP_2:

Note that the input to loop_2 are the adorned rules in the original adorned program.

From R0: q^{f}(y):- ancestor ^{bf} (a, y)

R02: q^{f}(y):- magic_R0_ancestor ^{bf} (a), ancestor ^{bf} (a, y)

From R1: ancestor ^{bf} (x, y):- parent (x, y)

N/A because there is no IDB in R1.

From R2: ancestor ^{bf} (x, y):- ancestor ^{bf} (x, z), parent (z, y)

R22: ancestor ^{bf} (x, y):- magic_R2_ancestor ^{bf}(x),
ancestor ^{bf} (x, z), parent (z, y)

Replace the original adorned rules with the above in P^{magic}.

P^{magic} after LOOP_2:

R01: magic_R0_ancestor ^{bf} (a)

R21: magic_R2_ancestor ^{bf}(x):- magic_ancestor ^{bf} (x)

R02: q^{f}(y):- magic_R0_ancestor ^{bf} (a), ancestor ^{bf} (a, y)

R1: ancestor ^{bf} (x, y): - parent (x, y)

R22: ancestor ^{bf} (x, y):- magic_R2_ancestor ^{bf}(x), ancestor ^{bf} (x, z), parent (z, y)

LOOP_3:

R03: magic_ancestor ^{bf} (a):- magic_R0_ancestor ^{bf} (a)

R23: magic_ancestor ^{bf}(x):- magic_R2_ancestor ^{bf}(x)

P^{magic} after LOOP_3:

R01: magic_R0_ancestor ^{bf} (a)

R21: magic_R2_ancestor ^{bf}(x):- magic_ancestor ^{bf} (x)

R02: q^{f}(y):- magic_R0_ancestor ^{bf} (a), ancestor ^{bf} (a, y)

R1: ancestor ^{bf} (x, y): - parent (x, y)

R22: ancestor ^{bf} (x, y):- magic_R2_ancestor ^{bf}(x), ancestor ^{bf} (x, z), parent (z, y)

R03: magic_ancestor ^{bf} (a):- magic_R0_ancestor ^{bf} (a)

R23: magic_ancestor ^{bf}(x):- magic_R2_ancestor ^{bf}(x)

Simplification:

1. Renaming all *magic...ancestor ()* to *magic()* since there is only one unique *magic* predicate.

   magic (a)

   q^{f}(y):- magic (a), ancestor ^{bf} (a, y)

   ancestor ^{bf} (x, y): - parent (x, y)

   ancestor ^{bf} (x, y):- magic (x), ancestor ^{bf} (x, z), parent (z, y)

2. Use magic (a) to bind corresponding ancestor parameters.

   magic (a)

   q^{f}(y):- magic (a), ancestor ^{bf} (a, y)

   ancestor ^{bf} (x, y): - parent (x, y)

   ancestor ^{bf} (a, y):- magic (a), ancestor ^{bf} (a, z), parent (z, y)

3. Remove all magic ().

q^{f}(y):- ancestor ^{bf} (a, y)

ancestor ^{bf} (x, y): - parent (x, y)

ancestor ^{bf} (a, y):- ancestor ^{bf} (a, z), parent (z, y)

4. Remove all adornment and yield the final rewritten rules.

q (y):- ancestor (a, y)

ancestor (x, y): - parent (x, y)

ancestor (a, y):- ancestor (a, z), parent (z, y)

Appendix F. Proposed New BNF for Cyrano Query Language

```
cyranoModel:    [objectList]
                ;
objectList:     object [objectList]
                ;
object:         [className] objectName [inSpec] [isASpec] [isDSpec]
                [withSpec] 'end'
                ;
 className:     objectName
                ;
objectName:     '(' objectName ')'
                | label
                | objectName SELECTOR1 label
                | objectName SELECTOR2 objectName
                | deriveExp
                ;
deriveExp:      label '[' bindingExpList ']'
                ;
bindingExpList: singleBinding [',' bindingExpList]
                ;
singleBinding:  objectName '/' label
                | label ':' objectName
                ;
inSpec:         'in' classList
                ;
isASpec:        'isA' classList
                ;
isDSpec:        'isDerived' classList
                ;
withSpec:       'with' attrDeclList [constraint]
                ;
classList:      className [',' classList]
                ;
```

```
attrDeclList:       'attribute' attrDecl [attDeclList]
                ;
attrDecl:           attrCategoryList attrSpecList
                ;
attrCategoryList: label [',' attrCategoryList]
                ;
attrSpecList:       attrSpec [';' attrSpecList]
                ;
attrSpec:           label ':' objectName
                ;
constraint:          'constraint' label ':' '$' formula '$'
                ;
label:              ALPHANUM
                    | LABEL
                    | '$' formula '$'
                ;
formula:            'exists' variableBindList formula
                    | 'forall' variableBindList formula
                    | 'not' formula
                    | formula '==>' formula
                    | formula 'and' formula
                    | formula 'or' formula
                    | '(' formula ')'
                    | literal
                ;
variableBindList: variableBind [variableBindList]
                ;
variableBind:       varList '/' className
                ;
varList:            ALPHANUM ',' varList
                    | ALPHANUM
                ;
literal:            FUNCTOR '(' literalArgList ')'
                    | ALPHANUM '(' literalArgList ')'
```

```
                | '(' literalArg infixSymbol literalArg ')'
                | BOOLEAN
            ;
infixSymbol:    INFIXSIYBOL
                | label
            ;
literalArgList: literalArg [',' literalArgList]
            ;
literalArg:     objectName
            ;
ALPHNUM:        (a-zA-Z0-9)+
            ;
LABEL:          (everything except: ' ' " " $ { } : ; , ! ^ - > = ( ) [ ] / blank)+
            ;
SELECTOR:       ! | ^
            ;
SELECTOR2:      => | ->
            ;
BOOLEAN:        TRUE | FALSE
            ;
INFIXSYMBOL: < | > <= | >= | = | <> | in | isA
            ;
FUNCTOR:        From | Label | To
            ;
```

Zhao-Ping Yu

Experience

Senior Member of Technical Staff / Member of Technical Staff (1/88 - present)

Alcatel Data Networks, Inc. Ashburn, VA, R&D
(Formerly Sprint International / GTE Telenet Communications Corp)

Current Project:
Frame Relay - ATM Inter-working Switch Systems.

Designed and Developed:
Frame Relay Switch Systems, ISDN Systems, QLLC PAD,
Frame Relay - X.25 Inter-working Switch Systems.

Software Engineer -Tesdata Systems Corp, Herndon, VA, R&D (5/85 - 1/88)

Designed and Developed:
Network Performance Monitoring and Analysis Systems.

Education

M.S. in Computer Science. (12/96-expected)
Virginia Tech - Northern Virginia Graduate Center, Falls Church, VA.

Graduate Study in Computer and Electrical Engineering. (9/86 - 5/87)
George Mason University, Fairfax, VA.

B.S. in Computer Science, B.S. in Mathematics, minor in Physics. (12/84)
College of the Ozarks, Point Lookout, MO. Graduated Summa Cum Laude.

Honors

Alcatel Data Networks Achievement Award, Alcatel Data Network Premier Employee Awards, Sprint International Champion Awards,
Who's Who Among Students in American Universities and Colleges,
Highest Scholastic Honor, Computer Work-Study Awards,
Kappa Delta Omicron Honorary Scholastic-Leadership Fraternity,
Hyer Scholarship, Scottish Rite Foundation Scholarship,
Rossmoor Foundation Scholarship, Karl Drexel Faculty Senate Scholarship.

Memberships

ACM, IEEE-CS.