THE UNIVERSITY OF CALGARY

A PROTOTYPE FRONT-END FOR A DECLARATIVE OBJECT - RELATIONAL DATABASE LANGUAGE EMPLOYING NATURAL QUANTIFIERS AND GENITIVE RELATIONS

BY

CARMEN-DANIELA RATA

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

FEBRUARY, 1995

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled: *"A Prototype Front-end for a Declarative Object - Relational Database Language employing Natural Quantifiers and Genitive Relations"* submitted by Carmen-Daniela Rata in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Dr. James Bradley, Supervisor & Chairman
Department of Computer Science

[illegible]

Dr. Graham Birtwistle
Department of Computer Science

[illegible]

Dr. Anton W. Colijn
Department of Computer Science

[illegible]

Dr. J. A. Rodrigue Blais
Department of Geomatics Engineering

Date 18th April 1995

Abstract

The Object Relational approach to OODBMS attempts to provide a sound mathematical foundation for the current trend in databases towards integrating object-oriented programming language facilities with relational database management. Almost all database languages for the object-relational approach are extensions to SQL, the standard relational declarative language. COOL is an SQL-like object-relational declarative database language designed for an extended NF2 data model. COOL introduces the concepts of the genitive relation and the natural quantifier. It is relational because it is based on the Genitive Relational tuple calculus and an Extended Relational Algebra (ERA). COOL is object-oriented primarily because of the object-orientation of its data model, and secondarily because of the object-orientation reflected in the language semantics and structure. This thesis describes the first implementation of a language of this type. The focus of the thesis is on the implementation of COOL as a prototype front end for a relational database system. The thesis also covers the design of extensions to COOL; these extensions for COOL are the data definition language and the inheritance mechanism. At the conceptual level, the prototype front end implementation comprises a three-layer architecture that maps COOL's object schema to SQL's relational schema. A two step translation of COOL declarative language expressions was chosen as the most promising approach for portability and flexibility for future development. This successful implementation of a prototype front end for COOL demonstrates the practical feasibility of the COOL approach to declarative languages.

Acknowledgements

I would like to thank my supervisor, Dr. James Bradley for his full involvement and help over the course of this research. The long discussions we had were very inspirational. Besides help with the thesis, he gave me good advice and helped me adjust to Canadian life. I am also thankful for his editorial suggestions, which substantially improved the quality of this dissertation.

I would like to express special thanks to Dr. Graham Birtwistle for his valuable advice and support during the writing of my thesis.

I would also like to thank Dr. Anton Colijn and Dr. Jon Rokne for their continuous support and concern.

I would like to express very special thanks to Robert Fridman for his unbounded availability whenever I needed some help with Sybase and other system administration problems. Many thanks also to John Aycock for his help and to Darcy Grant for his continuous support during my research.

I would also like to express many thanks to my colleague Fabian Gomes for his help when needed, especially in solving apparently unsolvable Latex problems.

I would like to thank my good friend Ying Liu for her continuous support and help.

I am deeply thankful to my parents for their intense support from far away, and for their love and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

To meet the requirements of new complex database applications, object-oriented database management systems (OODBMS) are emerging as attractive candidates. Database management systems based on the relational approach, which have dominated for more than a decade, have shown serious weaknesses when used with scientific and engineering applications. Complex objects, such as a document or an engineering design, and very large data sets, such as multimedia data, including video or sound, do not fit in the simple, tabular data structures of the relational model. A new database technology was clearly needed. Its goal is to combine conventional database features both with object-oriented concepts, and with the rich data type systems present in the object-oriented programming languages (OOPL).

The new database technology, called object-oriented (OO), has two main approaches. One is an evolutionary approach, or extended relational approach, that is based on the relational model extended to include OO concepts. The other is a revolutionary approach, or the OOPL approach, that extends OOPL with database functions such as persistence, shareability, transaction management and limited query facilities. It is well understood that from the application point of view, a declarative, or non procedural, database language by itself is not sufficient, since certain parts of an application are best coded in a procedural manner. The OOPL approach can be characterized as an attempt to provide a programming language with database query capabilities, that is, a unified programming and database language, in order to achieve efficiency in writing database applications. However, the OOPL approach has some serious limitations:

- It does not have the strong theoretical foundation of the relational model.
- There is not a single OO data model but rather a plethora of OO models.
- It does not provide a powerful standard declarative query language, comparable to SQL with relational databases, but rather a wide variety of query languages with limited capabilities.
- It lacks many traditional database systems features available with relational systems, such as automatic query optimization and processing, automatic concurrency control, dynamic schema changes, so on.

By contrast, the extended relational approach is an attempt to provide a sound mathematical foundation for an object-oriented approach to databases. It makes it possible for object-oriented features to be available to users of databases and at the same time preserves the major contributions of the relational model, such as powerful non procedural, or declarative, query languages.

Declarative languages in relational databases are based on first-order logic, due to the fact that the language needs to manipulate only individual elements of a domain. In an object oriented database we deal with non atomic domains, where an element can be a set, a relation or a list. As a result, a declarative language for an object-oriented database needs to be based on higher order logic. Several attempts have been made in this research area, but the problem of considering higher order logic in developing an OO declarative language is still open. However, in the practical world of applied declarative languages, there was a strong trend towards extended versions of the conventional relational declarative query languages, such as SQL and QUEL, as query languages for OODBMS. Declarative languages based on the extended relational approach are sometimes called Object-Relational languages [Sto]).

COOL (A Composite Object-Oriented Language) is an example of this type of language [Bra92a], and it was designed as an extension of SQL. COOL is innovative in the field of declarative languages by employing new concepts like genitive relations, and natural quantifiers [Bra93b]. Although COOL has a SQL-like syntax, its expressiveness is totally different. If in SQL we need to express queries in an entire-relation manner, in COOL we use an object-oriented manner. For example, with the relational approach, if one wants to know the names of the passengers of all Delta Airline flights from Athens to Rome, one uses constructs that deal with all of the Flight tuples (**flightno**, airline-code, aircraft-type, capacity) and all of the Passenger tuples (flightno, **passenger-record-no**, passenger-name, citizenship). Thus, one must think in terms of entire relations, that is, all of the tuples within all of the relations involved in the retrieval.

Suppose now with an object approach that a database involves flight entities (object class Flight {object-ID: integer, flightno: integer, airline-code: string, aircraft-type: string, capacity: integer, Passenger-list: set of passengers}) and passenger entities (object class Passenger {object-ID: integer, flightno: integer, passenger-record-no: string, passenger-name: string, citizenship: string}). The data for a specific flight forms a composite object instance, whose subobject instances form a hierarchical structure. Users seem to prefer to work with composite object instances when dealing with such a data base. As an example: *Retrieve all flights that have transported more than 100 passengers where most (or majority of) passengers have Canadian citizenship.* With SQL we would have to code something like :

```
select airline-code, flightno from Flight
where capacity > 100
and (select count (*) from Passenger
     where citizenship = Canadian
     and Passenger.flightno = Flight.flightno)
     >
     (select count (*) from Passenger
      where citizenship <> Canadian
      and Passenger.flightno = Flight.flightno)
```

The natural quantifier here **for most** is involved in this retrieval and must be implemented in SQL with the awkward count( ) construction above. When we think in terms of objects, as we usually do, we would have to code something like:

```
Retrieve each flight,
  with
    (a) capacity > 100, and
    (b) passengers ,
          most of whom have Canadian citizenship.
```

The above query can now be expressed in COOL as:

```
select airline-code, flightno from Flight
```

```
where capacity > 100
and for most Flight.Passenger-list*Passenger
                                (citizenship = Canadian)
```

or, using a natural language-like alias for the genitive relation *Flight.Passenger-list*Passenger* as *Flight's Passengers*:

```
select airline-code, flightno from Flight
where (capacity > 100
and for most Flight's Passengers (citizenship = Canadian))
```

No language of this type has ever been implemented before. My contribution and the focus of my research is the implementation of COOL as a prototype front end for a relational database system, as well as the design of extensions to COOL.

The implementation of COOL was done on top of Sybase DBMS. The extensions I designed for COOL are the data definition language and the inheritance mechanism. As an approach to COOL's translation, I have chosen to translate COOL expressions in two steps: (1) reduction of COOL expressions to Extended Relational Algebra (ERA) routines, and (2) translation of ERA routines into a set of SQL expressions. The translation of COOL into ERA routines offers flexibility for future development of a full OODBMS based on COOL, and the translation of ERA routines into SQL expressions offers portability of the COOL prototype, since it can be executed on any relational system that supports SQL. The translators were built using Lex and Yacc Unix compiler tools, adding further to the flexibility of the system, by making it easy to incorporate changes as COOL evolves in time.

The execution of COOL statements was validated by comparing the output of

COOL queries to the output of SQL queries executed on the back end relational engine. Not only did the comparison help to validate the COOL query processor, but it also demonstrated the conciseness of COOL queries compared with equivalent SQL expressions. Performance issues are not addressed in this work.

The implementation of COOL employs three levels of abstraction. Beginning with the highest, we have:

1. **Conceptual level.** This level uses a structurally object-oriented or semantic database model, such as the Entity Relationship model extended by ISA relationships and complex attributes.

2. **Extended relational or object-oriented level.** This level uses an extension of a non-first-normal-form relational model and is an equivalent representation of the E/R concepts of the previous level.

3. **Conventional relational or implementation level.** This level is used as a basis for the prototype implementation and is an equivalent representation of the previous two levels using the conventional relational model.

In effect, COOL's implementation approach can be regarded as a translation of COOL's object schema to the SQL's relational schema.

COOL can support inheritance and it has been demonstrated how this mechanism can be built for COOL. However, inheritance has not been implemented yet. Complex objects, methods and abstract data types for COOL are not yet fully investigated and their implementation needs further research.

I was strongly motivated to implement a language like COOL primarily because the language is easy to use (compared with SQL), easily readable, and is along the lines of a natural language. If one wants an oral communication facility for databases, an inevitable future development, COOL provides a natural way to do that.

The remaining chapters of the thesis are organized as follows. In Chapter 2, I discuss an evolution of data models in the database field, emphasizing the oscillation back and forth between instance-based and value-based models. In Chapter 3, the two main approaches to object-oriented databases are presented, together with the most important implementations of each kind. In Chapter 4 an overview of database languages is given. In Chapter 5 an overview of the COOL language as an object-relational declarative language that employs natural-like language features, is presented. In Chapter 6, a detailed presentation of the implementation design is given. In Chapter 7 we give a verification set of queries to proof the correctness of the implementation and we explain how the system can be used. Finally Chapter 10 contains concluding remarks and discussion of ongoing work.

Chapter 2

Evolution of data models

The evolution of database models can be seen as a continuous effort to offload tedious and repetitive functions from the application programs to the database system. This has made the task of the application programmers considerably easier but has generated a lot of research to increase the performance of database systems. The evolution of database models and systems can be compared with the evolution of programming languages from machine languages to assembly languages, and then to high level languages. The high level languages have certainly alleviated the task of implementing increasingly complex applications, but have required increasingly sophisticated compilers [Kim90].

A first comprehensive definition of a 'data model' was given by Codd, the father of the relational model [Cod81]. The concept of a data model thus essentially appeared together with the mathematical formalism of the relational model. As given by Date [Dat90], Codd's definition of a data model is a combination of three components:

1. A collection of data object types (essentially the entities that form the database structures).

2. A collection of general integrity rules, which constrain the set of instances of those object types that can legally appear in a database.

3. A collection of operators, which can be applied to such object instances for retrieval and other purposes.

At a higher level of abstraction, we can distinguish hierarchical database structures and network database structures. At this level, database systems can be categorized according to the data structures permitted. The network data structure can be handled by both CODASYL (presented later) and relational models. The hierarchical structure is a special case of a network, so CODASYL and relational models can both handle it. The hierarchical model essentially permits only hierarchical data structures.

According to the definitions in [Bra83] the hierarchical structure is the structure in which every entity (a record or an instance) of an entity set (or type) has at most one parent entity and for a parent entity there can be many child entities (a one-to-many relationship). The network structure is the structure in which at least one child entity has more than one parent entity (a many-to-many relationship).

These two database structure concepts have been central to the evolution of the database models and systems for the past three decades.

Over this period of time, the data model evolution can be viewed in terms of (increasing) power of operations or user commands versus the semantic complexity inherent in the model, as shown in figure 2.1.

Alternatively, data model evolution can be viewed as an oscillation between the hierarchical and network data structure orientation of the prominent database models, as shown in figure 2.2.

Ullman offers an **object-identity** point of view when discussing the development of database systems [Ull88]. He calls **object-oriented**, those systems that support object-identity and **value-oriented** or **record-oriented** those systems that do not. In this sense the hierarchical and network-model systems are object oriented. All

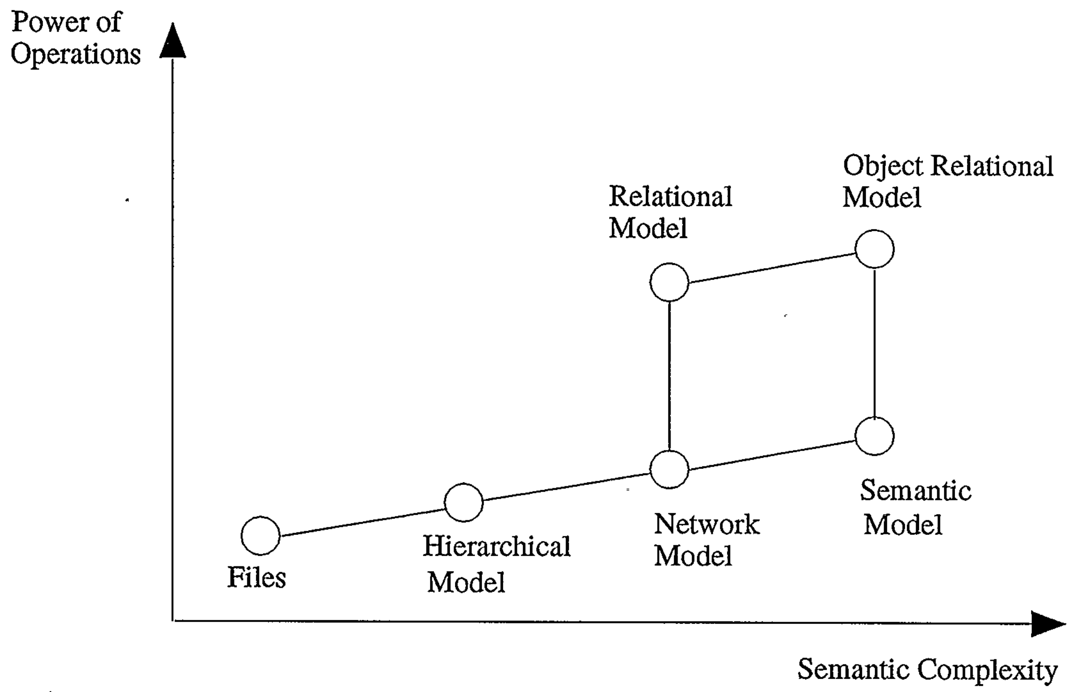

Figure 2.1: Evolution of data models

systems based on the relational model are value oriented, as are systems based on logic. However, one can simulate object-identity in a value-oriented system by use of unique codes for the tuples of a relation.

According to [Dat90], **the relational model** is the single most important development in the entire history of the database field. Thus, the history of the database field is divided into prerelational and postrelational eras. The most notable prerelational database systems fall into three broad categories (after [Dat90]):

1. Inverted list, such as the commercially available product CA-DATACOM/DB, from Computer Associates International Inc., which handles both network and hierarchical database structures,

2. Hierarchical, such as IMS(IBM) and System 2000(MRI), which handle only

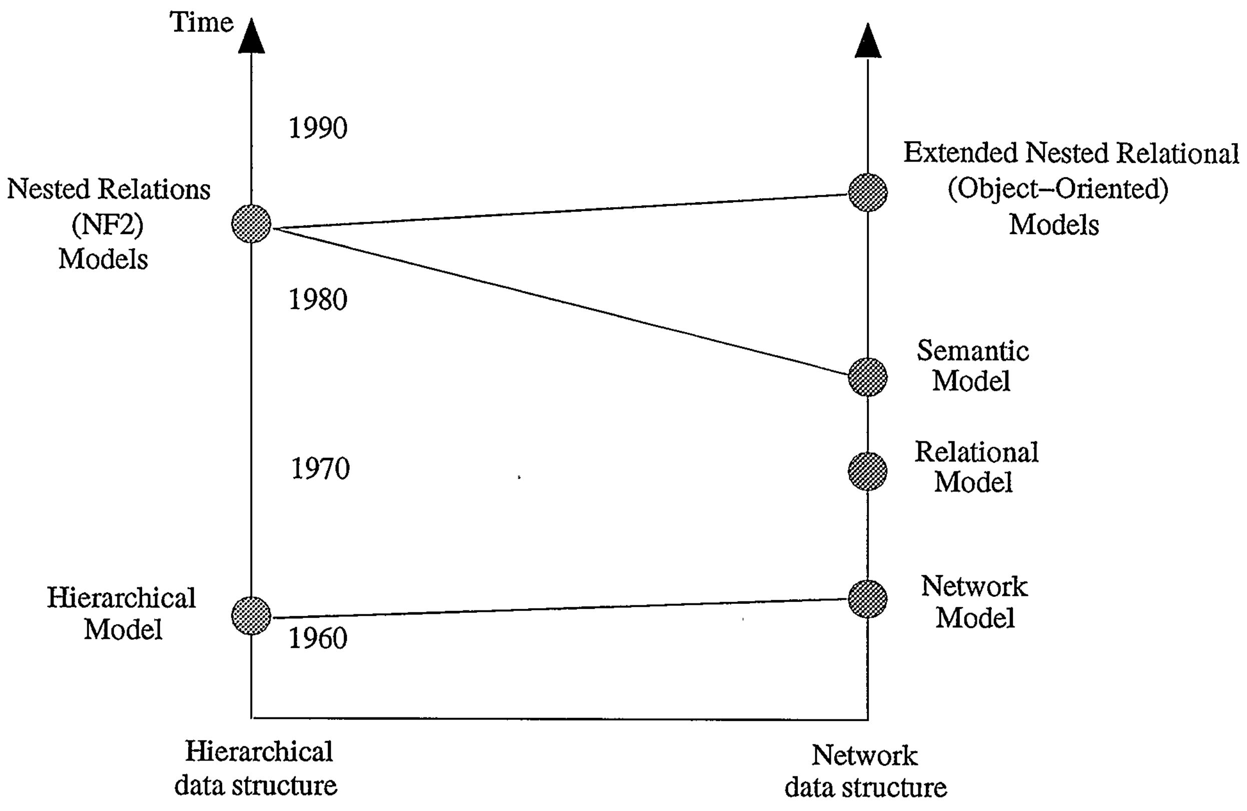
Time
1990
1980
1970
1960
Nested Relations (NF2) Models
Hierarchical Model
Extended Nested Relational (Object–Oriented) Models
Semantic Model
Relational Model
Network Model
Hierarchical data structure
Network data structure

Figure 2.2: Evolutionary oscillation of the prominent database models

hierarchical database structure,

3. CODASYL, such as CA-IDMS/DB, IDS, and TOTAL, which can handle network and hierarchical database structures.

Prerelational systems, often called 'record at a time' systems, accomplished the sharing of an integrated database among many users within an application environment. However, they lack data independence (explained later) and were basically programming systems that required a tedious programming or navigational access to the database.

The data models for the prerelational database systems implementations, hierarchical and network, were defined post facto, by a process of abstraction.

The relational model launched in 1970 by Codd arrived with many improvements and became the most implemented and most successful model of all time. Besides providing data independence, the relational model introduced for the first time the notion of declarative language into the database field.

Postrelational models started to appear in the late 1970s. They brought with them richer data types and operations necessary to meet the requirements of applications that were more demanding than the business data-processing applications for which the previous models had been developed. Some of the postrelational models were based on extensions of the relational model; others represented attempts at doing something completely different [Dat90].

The most notable postrelational data models are: the semantic model, the non-(first)-normal-form (NFNF or N1NF or NF2 or N2F) model and the object-oriented model. Each of the three post relational models are rather a class of models than a single model.

The major data models will be briefly described in the following pages.

2.1 The Hierarchical model

The hierarchical data model essentially handles only hierarchical database structures[1] The most widely used hierarchical database system is IBM's IMS. The design of the basic system dates from the late 1960s and is still in use today, it's replacement being very expensive. In IMS we can identify a conceptual and a storage schema that are

[1]They can handle some limited network data structures in a complex ad hoc fashion.

specified together in a DBD (Data Base Description) [Bra87]. The conceptual schema defines a **hierarchy type**. A hierarchy type consists of a number (sometime very large) of hierarchy occurrences or **data trees**. A data tree represents a database record and is a collection of file records or segments of different types. The segment from the top of a tree is called the root segment. The number of trees in a hierarchy type is equal to the number of root segments. We can easily identify a hierarchy with a composite object class and a tree with a composite object instance. If we add OIDs and the navigational access to data, we have an OO model.

IMS's most important disadvantages are:

- Duplication of segments in different database records and inconsistency because of that, and
- It does not offer data independence.

2.2 The Network model

If the hierarchical model is basically the abstraction of the IMS system of IBM, the network model is the abstraction of the CODASYL system. Heavily influenced by the COBOL programming language, CODASYL was developed by the Data Base Task Group (DBTG) [Bra83]. Two notions were introduced in the network model: **records** and **sets**. Records of the same type are grouped into distinct conceptual files (e.g.: collection of warehouses, collection of warehouse employees, so on). Also an additional grouping of records is done through the **owner-coupled sets**. An owner-coupled set groups records embodying a 1:n relationship. It contains a collection of owner-coupled set occurrence, that each contains a parent record and a collection of child records (e.g.: a warehouse with the employees who work in it). At the storage level, the parent file record contains a pointer to the first child file record. In the first child record there is a pointer to the next child file record and so on. In CODASYL terminology a parent record is called an **owner record** and a child record a **member** record.

The network model is object-oriented to the extent that it supports object-identity. Records of the network model have an invisible key, which is in essence the disk address of the record (currency indicator). More specifically, there is a currency indicator for each file in the data base, for each owner-coupled set, and a currency indicator for the whole database (current of run-unit — CRU indicator). The currency indicators support navigation through the data records when a retrieval is made.

In Ullman's opinion [Ull88] the CODASYL DBTG language and IMS's DL/1 language are object oriented database languages.

2.3 The Relational model

The fundamentals of the relational model were presented by Dr. E.F. Codd, a mathematician by training, in a classic paper [Cod70]. The mathematical formalism underlying the relational model is based on set-theory.

The first major relational products began to appear in the early 1980s. Since then the relational database systems (RDBMS) have dominated the marketplace with almost 200 commercial products [Dat90] running on just about every kind of hardware and software platform imaginable. Examples of such products include DB2, SQL/DS, the OS2/2 Extended Edition Database Manager, and the OS/400 Database Manager(from IBM), Rdb/VMS from DEC, Oracle from Oracle Corporation, Ingres from Ingres Corporation, and many others. The declarative relational database language SQL became an industry standard. The success of RDBMS was due to the simplicity of the relational data model. The SQL query language that came with it made a significant productivity enhancement in application development. Relational systems handle both network and hierarchical database structures.

The main concepts of the relational model will be presented in the following sections.

2.3.1 Formalization of relations

There are two definitions associated with the concept of relation: (1) the **set-of-lists** and (3) the **set-of-mappings.**

The **set-of-lists** definition of a relation is given by Ullman in [Ull88], as follows:

A **relation** is a subset of the cartesian product of a list of domains. Formally, a **domain** is a set of values. For example, the set of integers is a domain and so are the set of character strings, the real numbers, and the set {0,1}.

The **cartesian product** (or just product) of domains $D_1, D_2, \ldots, D_k$, written $D_1 \times D_2 \times \ldots \times D_k$, is the set of all k-tuples $(v_1, v_2, \ldots, v_k)$ such that v_1 is in D_1, v_2 is in D_2, and so on. For example, if we have $k = 2$, $D_1 = \{0, 1\}$, and $D_2 = \{a, b, c\}$ then $D_1 \times D_2$ is $\{(0, a), (0, b), (0, c), (1, a), (1, b), (1, c)\}$.

A subset of the product $D_1 \times D_2$, such as: $\{(0, a), (0, c), (1, b)\}$, is a relation. The empty set is another example of a relation. A member of a relation is called **tuple.**

Each relation that is a subset of some product $D_1 \times D_2 \times \ldots D_k$ of k domains is said to have **arity** k; another term for arity is **degree.** A tuple $(v_1, v_2, \ldots, v_k)$ has k components and is sometimes called a k-tuple.

A finite set of attribute names $\{A_1, A_2, \ldots, A_n\}$ for a relation, defines a **relation scheme.** If we name a relation 'r' and a relation scheme R with the attributes $A_1, \ldots, A_n$, we can write $R = r(A_1, A_2, \ldots, A_n)$.

The **set-of-mappings** definition of a relation is given in [Mai83], as follows.

A relation scheme R is a finite set of attribute names $\{A_1, A_2, \ldots, A_n\}$. Corresponding to each attribute name A_i is a set D_i, $1 \leq i \leq n$, called the domain of A_i. The domains are arbitrary, non-empty sets, finite or countably infinite. Let $D = D_1 \bigcup D_2 \bigcup \cdots \bigcup D_n$. A relation r on relation scheme R is a finite set of mappings $\{t_1, t_2, \ldots, t_p\}$ from R to D with the restriction that for each mapping $t \in r, t(A_i)$ must be in $D_i, 1 \leq i \leq n$. The mappings are called tuples. Thus it is possible to view tuples as mappings from the attributes names of a relation scheme to values in the domains of the attributes.

In the traditional view of a tuple as a list of values (ordered set), the tuples (Calgary, Underhill, 3615) and (Underhill, 3615, Calgary) would not be the same.

In the **set-of-mappings** view, we attach attributes names to columns of a relation, and the order of the columns becomes unimportant. Ordering adds nothing to the information content of a relation.

Since the relational database system allows the specification of columns of a relation in any order, the **set-of-mappings** definition of a relation is more appropriate. However [Ull88] there are situations such as when dealing with relational algebra, where we need to use the **set-of-lists** definition of a relation (e.g.: theta join needs the **set-of-lists** definition, semi join and natural join needs the **set-of-mappings** viewpoint). There is a trivial way of converting between the 2 viewpoints. Given a relation in the set-of-lists form, we can give attribute names to its columns, and it can be viewed as a set-of-mappings. Conversely, given a relation in the set-of-mappings form, we can fix an order for the attributes and convert it into a set of lists.

A relation can be viewed also as a table, where each row is a tuple and each column (or component) is called an attribute. The most important terms used in the relational model environment are summarized in table 2.1 [Dat90].

Table 2.1: Relational terminology

Formal relational term	Informal equivalents
relation	table
tuple	row or record
cardinality	number of rows
attribute	column or field
degree	number of columns
primary key	unique identifier
domain	pool of legal values

The relational model is based on the mathematical set-theory, as we have seen, but certain conditions are imposed on the concept of relation, as follows:

- In a relation duplicate tuples are not allowed and there is no order defined on the tuple.
- Attributes have no order, are referenced by names and must be unique within a relation.
- Attributes values are atomic.

These restrictions helped in defining a simple model for which a simple query language could be developed.

The best improvements brought by the relational model in the database world were :

- A declarative query language for building ad-hoc retrievals from the database, and
- The data independence, which is the ability to change the database schema (logical data independence) or the internal organization of data, such as indexes or record layout (physical data independence), without having to change the application programs.

2.3.2 Relational operators

There are two different kinds of notations for expressing operations on relations:

1. Algebraic notation, called **relational algebra**, where queries are expressed in a procedural manner by applying specialized operations to relations, and
2. Logical notation, called **relational calculus**, where queries are expressed in a declarative manner by writing logical formulas that the tuples in the answer must satisfy.

The relational algebra consists of a collection of eight operations that can be grouped in two categories [Dat90]: (1) the basic operations on sets that apply to relations: union, intersection, difference, and cartesian product, and (2) the special relational operations: select, project, join, and division.

The *basic operations* are:

1. **Union.** The union of relations R and S, denoted R $\bigcup$ S, is the set of tuples that are in R or S or both. We may only apply the union operator to relations of the same arity.

2. **Intersection.** The intersection of relations R and S, denoted by $R \bigcap S$, is the set of tuples that are in both R and S. We may only apply the intersection operator to relations of the same arity.

3. **Set difference.** The difference of relations R and S, denoted by R − S, is the set of tuples in R but not in S. We again require that R and S have the same arity.

4. **Cartesian Product.** With the cartesian product operation we have a special situation. In mathematics the cartesian product of two sets is the set of **all ordered pairs** of elements, such that the first element in each pair belongs to the first set and the second element belongs to the second set. The relational algebra version of the cartesian product of two product-compatible relations (they have no attributes in common) R and S of arity k_1 and k_2, respectively, is the set of all possible $(k_1 + k_2)$-tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S.

   The *special relational operations* are:

5. **Selection.** Select is a unary operator on relations. When applied to a relation R, it results in another relation that is a subset of tuples of R with a certain

value on a specified attribute. For example, we wish to construct a new relation NEWFLIGHT, consisting of tuples from relation FLIGHT for which the attribute Dest value is "San Jose". The algebraic statement can be written: NEWFLIGHT = select (FLIGHT (Dest = "San Jose")). The relation FLIGHT is the operand, **select** is the operator, and the relation NEWFLIGHT is the result. In the parentheses following the operand, we may have any logical expression involving operands that are constants or attribute names, relational operators $<$, $\leq$, $=$, $>$, $\geq$, $\neq$, and logical operators $\bigcap$ (AND), $\bigcup$ (OR), and $\neg$ (NOT).

6. **Projection.** Project is also a unary operator. Instead of choosing a subset of the rows as select does, project chooses a subset of the columns. The duplicate tuples are also removed from the result. E.g.: Project the Destination and arrival-time from relation FLIGHT. Result = project (FLIGHT (Dest, Arrives))

7. **Join.** Join, also known as **natural join** is a binary operator for combining two relations on some or all of their common attributes (or identically named columns). The natural join term is mainly used to distinguish the definition above from other join-like operations, such as: equi-join, theta-join, etc. The common attributes are also called join attributes. The result of a natural join operation between two relations r and s is a relation in which every tuple is a combination of a tuple from r and a tuple from s with equal values for their common attributes (the same with saying the intersection of their schemas). The formal definition is given by Maier in [Mai83]: "The natural join of re-

lations r on scheme R or r(R), and the relation s on scheme S or s(S), (the 'set-of-mappings' is needed here), with R $\bigcup$ S=T, is the relation q(T) of all tuples t over T such that there are tuples t_r from r and t_s from s with $t_r = t(R)$ and $t_s = t(S)$. Since $R \bigcap S$ is a subset of both R and S as a consequence of the definition $t_r(R \bigcap S) = t_s(R \bigcap S)$". Thus every tuple in q is a combination of a tuple from r and a tuple from s with equal $(R \bigcap S)$-values.

Other join-like operators, are:

(a) **Equi-Join.** Equi-join extends the join operator to handle comparisons between columns with different attributes names but equal domains. Comparisons, as in natural join are based on equality.

The main difference between natural join and equijoin is that natural join does not repeat connected columns.

(b) **Theta-Join.** Theta-join extends join to handle combinations of two relations on the basis of comparisons other than equality, as well (E.g.: $<$, $\leq$, $>$, $\geq$). Thus, equi-join is a special case of theta-join.

(c) **Semi-Join.** The semi-join of relation R by relation S, is the projection onto the attributes of R of the natural join of R and S.

8. **Division.** The divide operator has a rather complex definition, but it is quite useful in some situation. The definition is as follows. Suppose a dividend relation D with the attributes a and b, a divisor relation R with the attribute b, and a quotient relation Q with the attribute a. Then Q = D divide R such that for each a-value, a_i in Q, there exists in R a set of tuples $(a_i, b_1), (a_i, b_2), \ldots, (a_i, b_n)$

such that the set $b_1, b_2, \ldots, b_n$ equals R. We have not made use of this operation in this thesis.

Authors like Warden have proposed new operators of an algebraic nature to be added to the relational algebra set. Some of these are extend, summarize and generalized divide [Dat90]. This new operators mainly increase the computational capabilities of the basic algebra.

Codd also extends algebra [Cod79] to deal with nulls, and with 'outer' versions of union, intersection, difference, theta-join, and natural join. The 'outer' version of the natural join is called **outer join** and proves to be a useful operation. Outer join has been used in the implementation of COOL, the subject of the present work. As an extension of the natural join, the outer join brings into the relation resulting from the join, the tuples of one relation that have no counterpart in the other relation. The tuples, that otherwise were ignored, appear with nulls in the matching attribute positions. Some SQL systems (e.g.: Sybase SQL, standard SQL2) have implemented outer join.

2.3.3 Relational calculus

The algebra provides a collection of explicit operations — join, union, projection, etc. — that can actually be used to tell the system how to build desired relation from the given relations in the database. On the other hand the calculus provides a notation for defining the desired relation in terms of the given relations. Thus, if the calculus simply states what the problem is, the algebra gives a procedure for solving the problem.

Relational calculus is founded on a branch of mathematical logic called predicate calculus and is the source of declarative languages.

A fundamental feature of the calculus is the notion of the **tuple variable.** A tuple variable is a variable that ranges over some relation i.e., a variable whose only permitted values are tuples of that relation. Because of its reliance on tuple variables (and to distinguish it from the domain calculus), the original relational calculus was known as the **tuple calculus.**

2.4 Semantic models

Semantic modeling research appeared in late 70s and early 80s with the desire of capturing more of the meaning of data. Semantic modeling is used in conceptual schema design, and thus prior to a translation into one of the traditional models for ultimate implementation. Thus, semantic models have a higher level of abstraction compared with the relational model for example. Some of the best known semantic models are: Chen's Entity Relationship Model (ER) [Che76], the Functional Data Model (FDM) [Shi81] and the Semantic Data Model (SMD) [HM81]. A comprehensive survey of the semantic models can be found in [HR87].

2.4.1 Semantic concepts

I will present the basic concepts of semantic modeling using one of the most prominent semantic models, the Entity-Relationship (ER) model. The mapping of an ER model to any other model can be easily done, and one reason for this is the way in which the semantic concepts describe the real world. Thus, an ER schema consists of entity types and relationships interconnecting these types, along with attributes of both the entity types and the relationships.

The basic semantic concepts are:

1. **Entity**. The world is made up of entities but is quite difficult to define with precision what an entity is. A common definition for the entity is the one used in database circles [Dat90], that an entity is any distinguishable object that is to be represented in a database (E.g.: Person, automobile, Purchase order, Ship, Part, Department, Document).

   The notion of distinguishability of entities is close to object identity and the ER model, and semantic models in general, are regarded as object-oriented models [Ull88]. Essentially, semantic models encapsulate the structural aspects of objects, whereas object-oriented models encapsulate the structural and behavioral aspects of objects [Kin89].

2. An **entity set** (or type) is a group consisting of all similar entities (e.g.: all ships, all departments)

3. **Relationship**. Chen defines a relationship as 'an association among entities'. A formal definition is given by Ullman [Ull88]: **Relationship/ definition 1** — "A relationship among entity sets is an ordered list of entity sets. If there is a relationship R among entity sets $E_1, E_2, \ldots, E_k$, then the current instance of R is a set of k-tuples. Such a set is called a relationship set. Each K-tuple $(e_1, e_2, \ldots, e_k)$ in relationship set R implies that entities $e_1, e_2, \ldots, e_k$, where e_1 is in set E_1, e_2 is in set E_2, and so on, stand in relationship R to each other as a group".

K or the number of entity sets participating in a relationship is called the degree of the relationship. So, we can have a binary relationship (k=2), or a ternary relationship (k=3). The most common case in practice is k=2.

Relationships are classified according to how many entities from one entity set can be associated with how many entities from another entity set, into: One to one, one to many and many to many.

4. **Attributes and keys**: Entities sets and relationships have properties, called attributes. All entities in an entity set have the same attributes or properties. Each attribute takes values from a domain of values (e.g.: the domain of real numbers or character strings). Properties can be: simple or composite (the composite property 'position' might be made up of the simple properties 'latitude' and 'longitude'); key (an attribute or set of attributes whose values uniquely identify each entity in an entity set); single or multivalued (associates a set of entities to one entity, for example: the set of languages a person can speak); or base or derived (the value can be derived from other attributes).

5. **ISA relationship or Supertype/Subtype entities**. The concept of ISA relationship was not included in the original ER model [Che76] but was added later. Any given entity can be of several types simultaneously. For example some Aircraft are Helicopters and all Helicopters are Aircraft. Aircraft is a supertype and Helicopter is a subtype or there is a ISA relationship between Aircraft and Helicopter. We can also say that a Helicopter is a special kind of Aircraft and can inherit the attributes of Aircraft, but also has specific attributes such as number of rotors that do not make sense for other types of

Aircraft, such as fixed wing aircraft. We might find also that some Helicopters are Cargo and other are Passenger. So an entity subtype is an entity type and can have subtypes of its own, and the process can continue, generating a **type hierarchy**.

Type hierarchies are known also, as: **generalization hierarchies, specialization hierarchies** or **ISA hierarchies**.

2.4.2 Mapping ER into the relational model

If we map ER model into a relational model, an entity set maps into a relation, an entity maps into a tuple, and the key maps into the primary key of the base relation. In the relational context relationships can be also mapped to relations. However, according to the **relationship/ definition 1** only the many to many relationship qualifies for a true relationship, since it demands representation by means of a separate table. One to one and one to many relationships can always be represented by means of a foreign key in one of the participant tables.

A comprehensive presentation of different kinds of relationships and their representations in different data models is given in [Cat91].

Basically, the relational model and the object-oriented one are using the same techniques to represent relationships, that is, by means of attributes. The attributes could be simple (atomic values), or complex (reference, collections, or derived (functions)). The differences are: OO models use OIDs instead of using foreign key and primary key, and the relationships are more meaningful to the user (through the attributes used by the OO model) and cannot be corrupted so easily. We say that relationships cannot be easily corrupted when they are not associated with user-visible values, such as the foreign keys; all the values in the referenced object may be changed and the reference attribute still points to the same object. More details about relationships in the OO models will be given in Chapter 3.

Bradley [Bra92d] gives a thorough classification of relationships in relational databases, and the following definition of a relationship: **Relationship/ definition 2**, for tables — "A relationship R(A,B) between an arbitrary pair of relations (A,B) is defined by a relation r (a,b) containing the attributes a and b, the primary keys of A and B."

Relation **r** is obtained by a sequence of relational algebraic joins A*J*K* ...*B, followed by a projection on fields a and b, where J, K, ... are also relations. Since there may be many possible sequences of joins between A and B, it follows that many relationships between A and B may exist, each identified by a unique r(a,b).

In a relational database there can exist the following types of relationships: one to many (1:n), many to many (n:m), one-to-one (1:1), recursive 1:n, recursive n:m, and co-relationships.

A n:m relationship between two relations A and B can always be decomposed into a 1:n relationship between A and a third relation C, and a 1:n relationship between B and C.

The 1:n, n:m and the recursive relationships (either 1:n or n:m) can also be primitive (simple) or composite.

A relationship R(A,B) between two relations A and B is primitive if it is simply based on a single join between A and B, that is

$$R(A,B) \; = \; project \; ((A \; join \; B)(a, \; b))$$

(see **relationship/ definition 2**). A relationship between A and B is composite where there is no common join attribute in A and B, and the relationship is due to a series of joins involving a chain of primitive relationships, that is, involving other relations J, K, ... so on.

A relationship between A and B is called recursive if and only if A=B [Bra87].

Common recursive relationships are either simple one-to-many or many-to-many. Examples of recursive one-to-many are: (a) a relation whose tuples describe all the employees in a hierarchical business organization, where each employee except the president, reports to one other employee, and (b) a relation whose tuples describe corporations, where a corporation can be an entirely owned subsidiary of another. Common examples of recursive many-to-many relationship are: (a) the bill of materials where relations describe components and subcomponents for containment based objects, and (b)a relationship resulting from a relation whose tuples describe corporations, where any corporation can own part or all of the shares of other corporations.

2.4.3 Type constructors

An important form of abstraction in a semantic model is the type constructor. Type constructors are used for abstracting or building complex objects out of less abstract atomic types. There are two prominent type constructors: aggregation and grouping or association. From the set-theory point of view aggregation is fundamentally a cartesian product of a list of domains. An informal definition of aggregation is: grouping of different part types into a whole. There are two kinds of results when aggregation is applied: (1) an entity or object type result, aggregated from atomic attributes, the classical example being ADDRESS, which is an aggregation of STREET, CITY and ZIP, (2) a composite object result, aggregated from objects of different types, an example being a VEHICLE made of ENGINE, WHEELs, ELECTRONIC_DEVICES, and MECHANICAL_DEVICES. In the ER model aggregation is represented by a relationship.

Grouping or association is a constructed type, defined in [HR87] as a finitary powerset, used to built sets of elements of an existing type. E.g.: Sets of Languages or sets of Hobbies.

Some implementations of semantic models are worth mentioning, for example GEM [TZ84], and TAXIS [NCL+86], a project developed at the Univ. of Toronto, are both implemented as front-ends to a relational system.

2.5 The Non [First] Normal Form models

At the end of the seventies, Makinouchi [Mak77] proposed that the first normal form condition imposed by the relational model be abandoned, that is, the condition that attributes in a relation must be atomic. This idea was triggered by the need to model complex data objects with hierarchical structure, such as books, office documents, etc. A plethora of models were simultaneously proposed, such as the nested relational model [SS86], the Fisher and Thomas model [FT83], the V-relational model [AB84]. The same idea arose naturally in the context of semantic database modeling [AH84, HY84].

Non [First] Normal Form models are referred to with various names, such as nested relational, NF squared, NF2, NFNF, N1NF. There are small differences between them depending on the data structures allowed to occur as attribute values (records or sets, arrays, or lists).

The most famous NFNF model is the nested relational model [SS86] The idea of the nested relational model is very simple. Relations are allowed in place of atomic attributes. This hierarchical nesting of relations may be repeated for an arbitrary but fixed number of levels [SS89, SS86].

From a semantic modeling point of view, the construct **relation** corresponds to the application of one aggregation operation to construct tuples from atomic domains, that is the tuple constructor, followed by one association operation to construct a set of tuples, that is the set constructor. Complex objects have evolved from relations in that they are constructed by repeated application of tuple and set constructors.

Thus nested relations can be constructed by repeatedly applying the sequence aggregation-association operations to a collection of primitive objects or to a collection of composite objects, obtained by previous applications of the rule.

The best way to describe the nested relation model is perhaps by using a programming language (Pascal) like syntax [SS91].

```
type Department                          type Employee
  = set of record                          = set of record
      cno: integer,                            SIN: integer,
      name:string,                             name: string,
      budget: real,                            sal: integer,
      ...                                      ...
      staff: set of Employee,                  end;
```

```
    end;
```

In the example above, the relation Department is a nested relation, where the value of a 'staff' attribute is a set of Employee tuples.

Some systems based on NFNF models, are:

- Prototype implementations for an extended NF2 data model: AIM-P (Advanced Information Management Prototype) [PD89], AIM-II [DKA+86].
- The VERSO prototype, developed at INRIA, France [SAB+89] which was implemented in Pascal on Unix.
- The Darmstadt Database System (DASDBS) from The Technical University of Darmstadt [SS89]. It is based on the idea that no single DBMS could cover all the different needs of various new DBMS applications. Instead we can have a kernel with several front-ends, that form a family of database systems. The kernel integrates common features of a low level storage component, and allows efficient front-ends tailored to specific application classes.

An important point about nested relations and complex objects is that they are incapable of directly representing non-hierarchical or many-to-many relationships and inheritance. These capabilities are added using extensions of the model.

2.6 Functional models

The functional data model (FDM) is a semantic model based on explicit representation of attributes as functional relationships. FDM has proved to be a very important base for OO modeling. It was first introduced by Kerschberg and Pacheco in 1976

[HR87] and made best known by Shipman [Shi81]. Shipman developed an informal graph-based representation of FDM schemas and a data language DAPLEX, that is considered the first database access language to give an important role to attributes that are used as atomic types, or composite types (objects) built from atomic types. Thus, the functional model shares ideas with the OO approach, such as the navigational, or path-tracing, style of addressing objects that are functionally related to each other, and is called an object-oriented semantic database model [HR87]. Date [Dat90] considers the functional approach and the OO approach as being the same.

A well-known example of an OODBMS based on the functional approach is Iris from Hewlett-Packard Labs [FBC+87, Fis89, Bro91]. It is implemented on a relational system. The Iris data model is based on functional data models and languages, such as DAPLEX [Shi81] and GORDAS [EW81]. It has three constructs; namely objects, types and functions, and it supports inheritance, integrity constraints, complex objects, user-defined functions, and extensible data types. Objects may be referenced directly through their keys. The query language of Iris is called Object SQL (OSQL).

2.7 Object-Oriented Models

Object-oriented data models are the result of the emergence of new database applications classes [Cat91], such as:

- Computer-aided software engineering (CASE),
- Mechanical and electrical computer-aided design (CAD),

- Computer aided manufacturing (CAM),
- Office automation,
- Computer-aided publishing (CAP) and hypertext,
- Graphics,
- Scientific and medical applications,
- System services,
- Manufacturing and real-time control,
- Knowledge bases for AI and
- Business applications where traditional DBMSs have proven inadequate.

Although relational DBMSs have a firm theoretical foundation, these new applications have revealed many weaknesses in the relational model, and have focused attention on the need for:

- support for much more complex entities, such as design and engineering objects, and compound documents,
- abstract user-defined data types,
- the semantic concepts of generalization, aggregation, and association,
- temporal evolution of data, particularly the temporal dimension of data, and multiple versions or versioning of data,

- multimedia data such as audio, and video data,
- new capabilities such as manipulating stored complex objects and rule processing.

These needs, together with the need to reduce the cost of developing complex software systems, have brought the object-oriented programming paradigm into database technology.

The OO paradigm embodies abstract data types, encapsulation and inheritance which has made it easier to develop and upgrade applications. It should be remembered that ease of application development was the major driving force in the original evolution of database management technology from file systems to relational database systems.

OO database research, unlike other approaches to databases, started in a bottom-up fashion with the implementation of a number of working systems, without formalization of underlying concepts [Wie91]. Implementation of OODMBS was very active in the mid 1980's with projects such as Orion [KBC+89, Kim90], O2 [Deu91], and Iris [FBC+87]. A top-down movement started in 1989 when confusion about object database concepts made the need for a number of simple and clear rules obvious.

As in the case of NFNF models, there is not a single OO model, but rather a plethora of OO models. There is an attempt of standardization from the OMG group [Cat93] but the standard is not accepted by the vast majority of OODBMS developers [Kim94].

Two major manifestoes that give a 'general' definition of an OODBMS were written: [ABD+89] and [SRL+90]. These manifestoes define the two main trends in

OO technology, and they will be presented in Chapter 3.

2.8 Comparative views of database models

2.8.1 Relational versus OO models

The relational data model is value-based, as opposed to CODASYL, which could be called identity-based. The distinction arises from the mechanisms the data model provides for relating objects, a fundamental part of the modeling capability of any database system. A value-based model expresses the relationship between two objects by embedding the same value in the related objects. An identity based model can relate two or more objects independently of their embedded values. The relational systems are flexible and offer data independence by making a distinction between three layers: conceptual, storage and external.

OO data models are identity-based like network models such as CODASYL. They add a rich typing and extensibility. They add also encapsulation of data types and inheritance.

2.8.2 Semantic versus OO models

Some researchers refer to semantic models as being object-oriented, because they provide mechanisms for structuring complex objects (aggregation, grouping, relationships as attributes: constructors used to build software objects). So the distinction between the two sorts of modeling is not always well defined. King [Kin89] points out the differences between the semantic modeling and the object-oriented models, as follows.

- While semantic models attempt to provide 'structural abstractions', OO models provide 'behavioral abstractions'. Semantic models grew out of the same some sorts of concerns that inspired researchers in AI and knowledge representation. In contrast OO models were inspired by advances in programming languages. Semantic models are oriented toward the representation of data, while OO languages are concerned with the manipulation of data. While semantic models provide constructors for creating complex data types, OO models provide ways for embedding operations within data types.

- This distinction concerns the notion of inheritance. In a semantic model aggregations and attributes/relationships are 'inherited' down type hierarchies, that is, we have inheritance of structural components. In contrast, OO models focus on the inheritance of behavioral capabilities, in the form of the inheritance of operations embedded within types. Many of the OO systems have both structural and behavioral encapsulation facilities.

Chapter 3

Object-Oriented Trends

There appears to be a consensus among database researchers that next generation DBMS will be based upon the object paradigm. However, there is no consensus on a next generation OO data model.

The OO approach can be viewed as a combination of conventional database and object-oriented programming language (OOPL) technology. As a result of emphasizing either the database or the OOPL side, two main trends on OODBMS have emerged.

The **first trend** or the **evolutionary approach**, is to extend the relational model with a set of fundamental OO concepts (complex objects, abstract data types(ADT), access methods, and the encapsulation of data with methods), found in most object-oriented programming languages. The database language that embodies the united object-relational paradigm should be an extension to SQL. The database language should then be embedded in a wide variety of host programming languages.

Some of the OODBMS based on this approach are called object-relational (O-R) A definition of an object-relational DBMS is given in [Sto]. In this definition, an O-R DBMS should add the following OO concepts to a RDBMS: (1) unique identifiers, (2) user defined types, (3) user defined operators, (4) user defined access methods, (5) complex objects, (6) user defined functions, (7) overloading, (8) dynamic extendability, (9) inheritance of both data and functions (methods), (10) arrays. These are the OO capabilities of the Illustra or Montage (the commercial version of POST-

GRES) OODBMS. The same ideas of unifying relational and object-oriented data models are implemented in the UniSQL O-R DBMS [Kim92].

This type of OODBMS integrates well with existing relational databases and provides a smooth flow of data between engineering and business applications.

The **second trend** or the **revolutionary approach** is to extend object-oriented programming languages (notably C++ and Smalltalk), by allowing programming language objects to be persistent and sharable, that is, stored as a database, as well as permitting other database functions, such as transaction management and limited query facilities. The result is an object data model for which there exists no unique formal proposal but a variety of system-dependent data models.

The advantage of this approach is a single language for both database access and application programming. The disadvantages are (1) the lack of some essential database concepts, such as: a standard optimized query language, and mechanisms for concurrency control and reliability, and (2) the integration aspect of databases across several applications written in different languages seems to be lost since there are as many programming languages as OO database systems.

A common goal of OODBMS from both trends is to integrate applications programming and data management. The difficulty is that programming language environments and database systems are built on different concepts for typing, and computation. Typing systems in programming languages are rich, including: arrays, lists, ADT, as well as atomic types (integers, strings, etc), while typing systems in a database language typically include sets (relations) and atomic types. Computational models in programming languages are rich in manipulation capabilities, while computational models in a database language are more restricted but typically include search, insert, delete and update.

Basic OO data model concepts and annotated bibliographies are presented in many works, such as: [Kim90], [US90],[BM93], [Ban93], [Cat91], [Bro91], [Ull88], [ZM89a]. The two OO trends are best described in the Atkinson and Bancilhon et al., and in the Stonebraker et al. manifestoes that appeared in 89. Throughout the thesis I will use also the name "object-relational approach" for the first trend and "OOPL approach" for the second trend.

Since the system researched in this thesis belongs to the object-relational approach, we present the basic work in this field in the next section. In Appendix E, we review other work in the field of OODBMSs that we consider is helpful to an overall grasp of the field, but is not very relevant to the topic of the thesis.

3.1 The Evolutionary, Object-Relational Trend or Object Extensions of the Relational Database Model

3.1.1 Concepts

The Object Relational trend (O-R trend) is supported in the Third-Generation Database System Manifesto [SRL+90] issued by the Committee for Advanced DBMS Function, composed of Michael Stonebraker and other researchers from the field of relational database technology. The O-R trend is also outlined by Kim in the paper "On Unifying Relational and Object-Oriented Database System" [Kim92]. In the Third-Generation (TG) manifesto the first generation database systems are the older hierarchical and network database systems (CODASYL and IMS type systems or the systems of the 70s) and the second generation database systems are the relational

DBMS (the systems of the 80s).

The TG database manifesto is a set of basic tenets that should guide the development of third generation systems. The following covers the essentials of these tenets.

Tenet 1: Besides traditional data management services Third Generation DBMSs will provide support for :

1. **A rich type system.** A 'type' or a 'class' is a set of objects with similar structure and behavior. From a design perspective, objects model the entities used in the application domain. Each class has a name and a set of attributes that hold state values of the object and a set of operations (procedures and functions) that an object is subject to. Usually a 'type' and a 'class' are used interchangeably. Note that in the object model of the O_2 OODBMS [LRV89] 'type' and 'class' are used as distinct concepts. Further details about this original idea are in Appendix E.

   A rich type system includes an abstract data type system to construct new base types, and type constructors (array, list, tuple, set) that can be recursively applied to form complex objects. Prototype syntax for the above type constructors is contained in Starburst [GLPS91]. The above type constructors can be added to relational systems as natural enhancements. This approach has already been applied to SQL3 draft for the support of Abstract Data Types (ADT) and functions (more details will be given in Chapter 8).

2. **Single and multiple inheritance.** Inheritance is usually understood to be class inheritance, although inheritance can be also instance inheritance. Class

inheritance is the most common form of inheritance and is often called an ISA 1:1 relationship. The idea is that if Y inherits from X, Y is an X with some extra features. Class inheritance can be simple or single class inheritance, where the ISA relationships form a hierarchy, or it can be multiple class inheritance when ISA relationships form a network.

Single inheritance is essential but multiple inheritance is necessary to cover all the situations that can occur, so that in general the inheritance relationships form a directed graph. Multiple inheritance is optional in the [ABD+89] manifesto that supports the OOPL approach.

3. **Functions** (database procedures or methods) **and encapsulation.**

   Encapsulation refers to coupling of specific methods (operators, functions or procedures) to classes of objects. Thus, an object encapsulates both state (value of the object) and behavior (set of methods). Through the implementation of methods encapsulation provides data independence, allowing the private portion of an object to be changed without affecting applications that use the object class.

   Encapsulation has the advantage of encouraging modularity, but a total encapsulation can make some data elements inaccessible. For example the only way to access the Employee class might be to execute a function call, e.g. Hire(Employee). This is a restriction that ignores the needs of the query language to have access to each element directly. The query language should have access to the data elements inside data types. It is for this reason that some OODBMSs drop the encapsulation restrictions of access to the private implementation of an object when a query language is used, but keep it for access to object classes from application programs. This is called partial hiding. Of course, encapsulation can be avoided by defining methods for all the attributes that have to be visible to the query language. Other techniques can be used [Cat91].

Functions should be written in a high level nonprocedural language (embedded via a preprocessor in the query language) and have DBMS access through queries. Thus, functions should execute query expressions and not perform their own navigational programming using calls to some lower level DBMS interface as in prerelational systems.

4. **Unique Identifiers (OIDs).** OIDs for records should be assigned by DBMS only if user-defined primary keys are not available.

   If a primary key exists and is known that will not change (SSN, student ID number) we can keep it because it has a natural, human readable meaning.

   In the opinion of the writer an extra UID is still necessary to assure the uniqueness of an object instance. Many problems can arise with the primary keys (or using descriptive data for identity), and they are discussed in [KC89]. The solution is support for a system-generated identifier for objects that is independent of their external descriptive data, so that the system can preserve an object's identity when object sharing and updating are performed, as well as within complex objects, regardless of changes in data or structure.

Tenet 2: Third Generation DBMSs must subsume second generation DBMSs. Thus they should keep the major contribution brought by the relational model, such as non-procedural access and data-independence, and support of updatable views.

In the Atkinson et al. [ABD$^+$89] manifesto, the non procedural access, or the ad hoc query facility, can be of any convenient form. By contrast, the TG manifesto emphasizes that essentially all programmatic access to a database should be through a non-procedural language.

The access by a query language will be either by adding query language constructs to multiple persistent programming languages or by embedding a query language in conventional programming languages (which is the current approach).

Tenet 3: Third Generation DBMSs must be open to other subsystems, and that is:

1. They should be accessible from multiple programming languages by employing a closer match between the data type systems of the database and the programming language, and allow that any variable in a user's program to be optionally persistent.

2. Persistent programming languages supported on top of a common DBMS by compiler extensions and a (more or less) complex run time system would be a good idea.

3. The query language should be an extension of SQL, the uncontested standard. Additional query languages might be developed, for specific applications.

In conclusion, the object-relational approach combines traditional database theory concepts with programming languages concepts instead of attempting to incorporate database concepts into a programming language.

3.1.2 Examples of models and implementations

The first extensions made to a relational data model were semantic extensions, and were object-oriented in concept. The common goal of these extensions was to bring the relational data model closer to the real world since unnormalized relations more closely resemble the entities of the real world. Among the first semantically extended relational models were:

- **Extensions of NF2 data models.** Scholl and Schek [SS91, SS90] have developed an object model, called 'relational object model' by extending a NF2 model and using concepts from KL-ONE, a knowledge representation model used in AI and relational database model (see also the NF2 data model from Chapter 2). They have added an entity generalization/specialization facility (inheritance) and network data structure support to the NF2 data model. The network data structure support is achieved by functions as abstractions of both attributes and relationships. These 'functions' map instances of a domain type to either an instance or to a set of instances of another domain type. Thus, a classic attribute can be viewed as a mapping in atomic domains. The functions can play the role of (1) pointers from programming languages, (2) of reference attributes [Cat91], (3) of OIDs from various object-oriented database models, or (4) of the foreign key from the relational model. This technique of using functions as attributes makes it possible to extend NF2 relations from a hierarchy of relations to a network of relations.

  The data model of IRIS system [Bee88] and OODAPLEX [Day89] (the OO extension of the FDM) use the same kind of object-function-model.

The example presented in Chapter 2, in the NF2 data models, now becomes:

```
type Company                        type Employee
  = set of record                     = set of record
      cno: integer,                       SSN: integer,
      name:string,                        name: string,
      budget: real,                       sal: integer,
      president: Employee,                works_for: Company,
      staff: set of Employee              owns: set of Vehicles
      end;                                end;
```

The non atomic types : **staff**, **works_for**, **owns**, in the type definition are expressed as functions.

For example, **staff** is a function that maps **Company** to a set of objects with the type **Employee**. Also, the pair of functions (staff, works_for) describes a one-to-many relationship.

- **The POSTGRES system.** POSTGRES is the most powerful implemented prototype of this trend [Cat91]. POSTGRES has been under construction since 1986 at the University of California at Berkeley [SK91, Sto87, MR86, Cat91] and as the name shows is a follow-on to the INGRES RDBMS. POSTGRES has a recent commercial version called Illustra. The data model of POSTGRES [RM87] is an extension of the relational model with:
  - user defined Abstract Data Types (ADT) and associated operators,
  - the structured attributes *type procedure* and *array*, and

- data and procedure inheritance.

POSTGRES extensions provide support for semantic and object-oriented concepts such as: aggregation, generalization and association, and complex objects with shared subobjects.

The class in POSTGRES, can be associated with a relation and is a collection of instances of objects (or tuples). A class has attributes of fixed type that can be atomic or structured. Each instance has a unique system generated identifier (OID), which is readable but not updatable by the user. Primary keys for instances (or tuples) can be optionally defined .

There are three kinds of classes: (1) real or base classes, whose instances are stored in the database, (2) derived or view (or virtual) classes, whose instances are not physically stored, (3) version of another class which is stored as a differential relative to its parent class.

POSTGRES contains an extensive type system. Important types are:

- **new base types** built using ADT definitions, for example a definition of an ADT that represents boxes:

```
define type box is
           (InternalLength=16,
            InputProc = CharToBox,
            OutputProc = BoxToChar, Default='' ``
            )
```

A box is represented as a character string that contains two points that represent the upper-left and lower-right corners of the box. CharToBox is a function that takes a character string that represents a box and returns a 16 byte representation (4 bytes per x- or y-coordinate value. BoxToChar os the inverse of CharToBox.

– **arrays of base types.** For example if an employee receives a different salary each month, we could use:

```
retrieve (EMP.name)
        where EMP.salary[4] = 1000.
```

– **composite types** that allow the construction of complex objects (aggregation) with a hierarchical internal structure. There are two composite types:

1. indicated by **class name**, and containing zero or more instances of that class, for example, the class EMP contains instances of type EMP for the type of the attribute 'manager'.

```
create EMP (
            name = c12,
            salary = float[12],
            age = int,
            manager= EMP
           )
```

2. indicated by **set**, whose value is a collection of zero or more instances from all classes, for example:

```
add to EMP (hobbies = set)
```

The elements of a composite type are addressed by nested dot notation or path expressions, present also in IRIS, ORION, O2 and Gem, such as:

```
retrieve (EMP.manager.age)
where EMP.name=''Joe''
```

EMP.manager.age gives the age of a manager from class EMP, and EMP.name gives the attribute name of an instance from class EMP.

- **type procedure.** There are three kinds of procedures (or functions) in POSTGRES:

  1. C functions (whose arguments are base types or composite types and are dynamically loaded when used in a query),
  2. operators (functions of one or two operands which are base types; for example: operator '!!', returns 'true' if two polygons overlap),
  3. POSTQUEL functions (any collection of commands in POSTQUEL – the query language of POSTGRES – can be defined as a function).

POSTGRES can be called from many different languages.

- **The Starburst system.** Starburst project is another major project illustrating this trend at the IBM Almaden Research Center, initiated in 1985. Starburst is an extensible DBMS prototype based on the relational model and on extensions of SQL [LLPS91, Cat91, LH90]. It is a prototype that was not built on the foundation of any earlier system. The extensions to relational technology are also different from those in POSTGRES, where the relational model was extended. The extensions in Starburst are present at the DBMS component level. We have (1) storage and access method extension, (2) query analysis extensibility, (3) query optimizer extensions, (4) query language extensions, for example: recursive queries, table expressions and table functions; (5) complex object support, and (6) type extension mechanisms.

The extensions developed for Starburst, incorporate the best features of many existing data base technologies, such as:

- solid theoretical foundation and a declarative query language (an extended form of SQL) that can be optimized,
- a richer type system, enhanced performance using system maintained pointers to related objects, encapsulation of behavior with the data, object identifiers for stored objects, large structured complex objects (an application program interface has not yet been implemented for them) and support of hierarchies of user defined types and functions (features retained from the object-oriented world),
- user defined rules to respond to changes in the database (active database) and, general recursion added to SQL.

Starburst has five kinds of functions:

1. **Scalar functions.** These have one or more scalar arguments and produce a scalar, such as: trigonometric functions or the square root function,

2. **Aggregate functions.** These operate on an entire column of a table and produces a scalar output, for example, AVG() in SQL,
3. **Set predicate.** This special case of an aggregate function returns a boolean value,
4. **Table functions.** These are functions that produce a table as output. They can have scalar inputs, or table inputs. Table functions are useful for importing data from outside the database and presenting it as a view or virtual table (e.g.: the Unix 'ls' command is implemented as a 'table function' whose parameter is the directory to be searched).

Unfortunately user-defined functions must be linked with the rest of Starburst, which is currently an inconvenience.

The type system for Starburst is still under design. Eventually Starburst will support an extensible , hierarchical type system in which all user-defined types can be encapsulated. Encapsulation is performed differently from the OO paradigm. In order to access a type a user need to have privilege access to use a function 'unwrap'.

.Single and multiple inheritance of data and functions will be also supported.

The set of quantifiers for Starburst query language will include the universal quantifier ALL and MAJORITY.

Complex objects in Starburst are stored using two approaches:

1. The entire complex object can be stored in a 'long field' in which representation of the object is entirely under user control. A selection predicate

thus cannot be applied.

2. A complex object's atomic components can be stored as rows in tables, and the object can be constructed by composing these rows using a new kind of relational view, an extension to Starburst, called eXtended Normal Form (XNF). This has not yet implemented.

- **Unified relational and object-oriented data model.** Kim is also in favor of the approach that an OO model can be viewed as an extended relational model and has introduced an equivalent of this model called 'unified relational and object-oriented data model' [Kim92, Kim]. Kim is also in favor of a declarative query language that can be optimized. Kim's data model was used for building a commercial database system, called UniSQL [Kim92].

Chapter 4

Database Languages

The core of a database system is its database language. A database system is in essence software that implements all the functions supported in a database language. A database language is an embodiment of a data model and a database model is the foundation of any database system. From this perspective, a database language is a very important component of a DBMS.

In principle, every database language has at least two component sublanguages:

1. **Data Definition Language (DDL).** This is a language that specifies the conceptual scheme. It is rather a notation that describes the types of entities and relationships among types of entities, in terms of a particular data model.

2. **Data Manipulation Language(DML).** This supports the manipulation or processing of objects by operations performed on the database. DML it is also called 'Query Language' (QL) but more correctly the query language performs only retrievals and it is included in the DML. Besides retrievals (or queries) a DML executes update, delete, so on.

Besides these two sublanguages, there can be a third sublanguage called a **data control** language that allows the database administration.

Often the manipulation of the database is done by an application program. The programs that manipulate databases are written in a so called **host language**, which is a conventional programming languages such as: C, Pascal or COBOL.

4.1 Query Languages

The most common query languages are the relational ones. The relational model offered us two types of QL: the procedural relational algebra and declarative or non-procedural relational tuple or domain calculus. Procedural and declarative database languages are equivalent in expressive power. Relational calculus is a straightforward adaptation of the first order predicate calculus. Relational algebra can be viewed as a functional language. It consists of a fixed collection of operations: union, difference, product, selection, projection, join, intersection, as described earlier. We have no predicates as we have in the calculus, and the relations are simply named constants.

4.2 Declarative database languages

"A declarative language is a language in which one can express what one wants, without explaining exactly how the desired result is to be computed" [Ull88].

The relational model has been very successful to a large extent because of the declarative languages, such as SQL and QUEL, that it introduced.

Declarative languages bring with them ease of use, associative data access and optimization. Essentially they have moved the optimization of access from the user to the database system. Thus, other factors being equal, users prefer declarative languages [Ull88].

The opposite of a declarative language is one in which we give the steps that lead to the desired result. This kind of language is called procedural (e.g.: Fortran, Pascal and C are procedural programming languages, and relational algebra is a procedural database language).

However, although the emergence of the OO approach to database systems has increased data modeling capabilities, it has been accompanied by a move back to procedural query languages. This has shifted the access optimization problem back to the user.

The need for declarative languages is currently the subject of much debate in the research community. The question "Do we really need declarative languages?" has often been discussed. Unfortunately, declarative languages cannot support all data processing needs. They are not computationally complete, so that they often need to be embedded in host languages. This leads to many problems. One of them is the annoying 'impedance mismatch' that occurs between the different data models embodied in both the query language and the programming language. This is a major source of confusion for application programmers, and it is why a stated primary goal of some OODB developers is to support an integrated application development language rather than an ad-hoc query language [ABD+89].

However, many applications involving business, and scientific databases need declarative language support. Thus, the advocates of the OO trend supported by the Committee for Advanced DBMS Function [SRL+90] insist that "we should not give up the benefits of declarative languages".

Overall, however, it is generally agreed in the OO database community that declarative languages are an integral part of the definition of a data model and that they must co-exist in the same system with procedural languages.

4.2.1 A logic approach for OODBs Query Languages (QL)

Many features included in OODB involve higher order logic concepts. Queries may involve quantification over sets and relations. Functions are involved in inheritance mechanisms, and, together with complex structures, are treated as data.

In [Bee90], Beeri tries to provide a logic-oriented modeling for OODB. Logicians distinguish between first order and higher order logics. Individual elements of a domain are considered first order. More complex constructs, such as sets, relations and functions are higher order. First order allows one to 'manipulate' only individual elements. Relations and functions are allowed as higher order constants in the first order but higher order variables are disallowed.

Higher order logic allows manipulation of higher order constructs. For example, second order predicate calculus allows quantification over relation-valued variables.

To summarize, in first order logic only individual elements of domains are considered as data, and functions and relations exist only as schema level elements; in higher order logics, sets, functions, and relations may be data as well.

The reason why first order predicate calculus is usually used, rather than some more expressive higher order variant, is that it has a sound and complete axiomatization, that is, the denotations of 'provable' and 'true' coincide, and problems are algorithmically tractable. This justifies its use for expressing queries and integrity constraints. This property fails for even second order logic. For instance in the second order logic, quantifiers can range over all the second order values that can be constructed from given atomic domains, and this leads to intractability. However, if the quantifiers are restricted to range only over restricted sets of second order values, the resulting calculus may be more tractable.

In the database world there is an interest in what is given in the database rather than in what could potentially exist. Therefore it has been of interest to both logicians and computer scientists to develop restricted higher order logics. The study of restricted higher order logics is an important research area in the theory of OODB and is a prerequisite to the development of a formal OO data model [Bee90].

Several practical attempts have been made to incorporate restricted higher order constructs into functional and logic programming, for example, HiLog [CKW92], and F-logic [KL89]. Nevertheless, it is still unclear whether or not the use of restrictions is a good solution to the problem of higher order programming in OODB.

A less formal discussion of the features of the QL for OODBMS, in an attempt to create a unique framework for analysis of QL is presented in [BNPS92].

4.2.2 Relational declarative query languages

Declarative languages in relational databases have their source in first order predicate logic on which the relational model is based. Important relational query languages are:

- DSL ALPHA, which is based on relational calculus. DSL ALPHA was developed by Codd at IBM. It is often used as retrieval standard against which the retrieval power of other query languages may be compared. An arbitrary relational query language L is said to be relationally complete if L expressions may be used to specify any retrieval that may be specified by DSL ALPHA expressions [Cod72].

- SQL (Structured Query Language) was developed as part of the SQL project at IBM and was later incorporated into the System R prototype. The language is relationally complete, that is, it may be used to specify any retrieval that can be expressed by DSL ALPHA. From a user standpoint, SQL is probably the most important relational retrieval language at this time.

- QUEL is the query language of INGRES, a relational DBMS developed at Berkeley. QUEL most closely resembles relational calculus. It is not widely used.

4.3 Query languages in the OO approach

Several declarative languages for object models have been offered. Query languages for OO systems can be classified into five broad categories according to the approaches on which they are based.

1. Declarative query languages based on Higher order logic approaches.

   Examples are F-logic [KL89], HILOG [CC89], and Noodle [MR93].

2. OO extensions of functional languages.

   Some examples of this type are: O_2 query [Deu91], LIFOO a functional query language for O_2 [BLM91], OODAPLEX [Day89].

3. Query languages as persistent object-oriented programming languages.

   These languages either extend Smalltalk or C++. OPAL (GemStone) extends Smalltalk. The query languages of ONTOS, ObjectStore, Objectivity, and

Versant OODBMSs, and the language ZQL[C++] [Bla93] are based on C++. A survey of the database languages for new generation database systems is available in [Las92].

4. Object-Oriented extensions of existing relational declarative and procedural query languages. This is probably the most important category and we consider it in some detail.

4.3.1 Object-oriented extensions of relational languages

These are extensions to tuple calculus languages, such as SQL, and to domain calculus languages like QUEL.

- a) Extensions of SQL or SQL-like languages. Some examples are:
  - COOL (Composite Object-Oriented Language). COOL is a declarative database language designed for an extended NF2 data model [Bra93a]. Thus, COOL is based on the object-relational approach to object-oriented declarative languages. It introduces the concepts of genitive relation and natural quantifiers. A presentation of the language is given in Chapter 5. The implementation of COOL, which is the main topic of this research, is presented in Chapter 6.
  - OSQL in IRIS OODBMS. The query language is Object SQL (OSQL) [Bee88, WLH90], which combines an SQL like syntax with a functional and semantic style. The functional style is present in the definition of attributes that are single argument functions. This style also shows in the definition of relationships between object types by means of functions.

The query processor translates Iris queries and functions into an extended relational algebra format that is optimized, and then executed by a storage manager which is conventionally relational.

– SQL/X. This is the query language of the commercial object-relational DBMS UniSQL [Kim92]. SQL/X allows path queries (like in Gem and POSTQUEL), queries against nested classes and hierarchies of classes, queries that include methods as part of search conditions, and queries that return nested objects. SQL/X is an upwardly compatible extension of SQL.

– Starburst query language. The Starburst query language [LH90] extends relational algebra and supports user-defined extensions to query analysis, optimization, execution, access methods, and storage methods [Cat91].

– ORL. ORL [UhCLS94] is an object retrieval language and has been implemented on top of ONTOS OODBMS.

- b) Extensions of QUEL. The main example is POSTQUEL [RM87]. POSTQUEL is the set-oriented query language of POSTGRES and is based on QUEL language of INGRES DBMS. It supports user defined functions and operators, arrays, path expressions, and inheritance.

4.4 The future of SQL

Since being adopted by both ANSI and ISO as a standard language for database access in 1986, SQL has gone through several revisions. The last revision is SQL-92 which provides increased functionality [Kul93] by including additional data types and explicit data type conversions, the concepts of information schema, domains, temporary tables, additional join operations such as outer join, new cursor options, and facilities to drop or alter schema objects. A new version, SQL3 upward compatible with SQL-92, is expected to appear in 1995-1996.

Two major shortcomings of SQL-92 addressed by SQL3 are: the lack of a rich type system required by complex applications, and the lack of computation completeness. To address the first shortcoming, SQL3 incorporates an extensible object-oriented type system borrowed from object-oriented programming languages. The SQL3 draft supports concepts such as: objects identity, ADT (abstract data types), user-defined functions, encapsulation, single and multiple inheritance (subtypes-supertypes), polymorphism, dynamic binding and rules [CMCG94]. The second shortcoming is addressed by the addition of procedural language constructs.

SQL3 can be thought of as an object-oriented programming language with built-in support for collection types (multisets or tables) and non-procedural query facilities.

Chapter 5

COOL and Extended Relational Algebra

5.1 Overview of COOL

COOL is an **object-relational declarative** database language.

COOL is **relational** because it is based on Genitive Relational tuple calculus [Bra92a], and an Extended Relational Algebra (ERA) [Bra94]. The Genitive Relational tuple calculus is an extension of relational tuple calculus and has both relation names and derived genitive relation names serving as implicit tuple variables. The Extended Relational Algebra is an extension of the conventional relational algebra with specific operations for reducing the natural quantifier expressions. ERA will be presented later in the chapter. Thus, COOL is soundly based on the set-theory.

COOL is **object-oriented** primarily because of the object-orientation of the model, which is an extension of the relational data model, and secondarily because of the object-orientation reflected in the language semantics and structure.

A simple way to demonstrate COOL's object orientation is by comparing it to SQL, a genuine relation-oriented language [Bra93b]. For example consider a database about parks. A Park can contain Forests and a Forest can have Campgrounds. A relational (Bachman) diagram of the database is shown in figure 5.1.

Take for example the retrieval from [Bra93a]: *"Retrieve full data on each California park where all of its forests exceed 10 square miles and have only campgrounds with fireplaces"*.

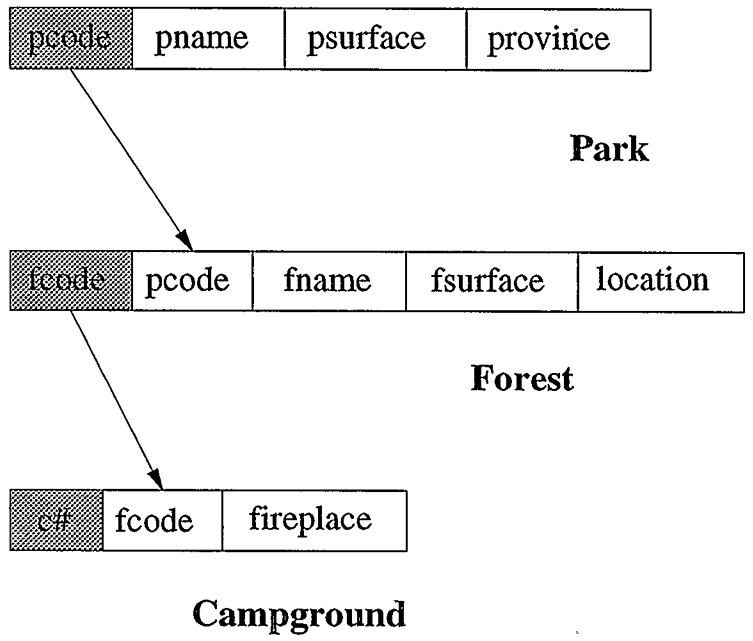

Figure 5.1: Parks Database 1

The SQL expression is:

```
select * from Park
        where location = ''California''
        and pcode in (select pcode from Forest)
        and pcode not in (select pcode from Forest
                          where fsurface <= 10
                          or fcode in
                             (select fcode from Campground
                              where fireplace = ''no'')
                          or fcode not in
                             (select fcode from Campground))
```

The COOL expression is:

```
select * from Park
where location = ''California''
and for all Park's Forest
               (fsurface > 10
                and for all Forest's Campground
                               (fireplace = ''yes''))
```

The above example deals with a hierarchical database structure, that defines composite objects (or aggregations).

The object orientation of COOL (see example above) is supported by the higher level of abstraction implicit in the specification of the relationships between the entities (a park has many forests and a forest has many campgrounds). Thus, in COOL we think in terms of objects only (a Park has Forests and Forests have Campgrounds). By contrast the SQL query is expressed in terms of entire relations (all the Forest tuples, all the parks tuples). Thus, in SQL we think in terms of entire relations and need to make sure that we join the relations on the right fields. We also need to correctly apply rules and logical operators, such as de Morgan rules, and double negation for the universal quantifier; in addition the natural quantifiers cannot be used. All these relation-oriented semantics for a language are error prone, require a logical mind set and take quite a long time to learn to use.

In contrast, COOL is based on an object-oriented data model, and offers an *object-oriented* approach to writing queries.

If in the example above, the quantifier **for all** is replaced by the quantifier **for most**, the COOL expression would remain the same with the exception of the quantifier:

COOL:

```
        select * from Park
        where location = ``California''
        and for most Park's Forest
                            (fsurface > 10
                             and for most Forest's Campground
                                             (fireplace = ``yes''))
```

However, the SQL expression needs to be changed to:

```
select * from Park
        where location = ``California''
        and (select count (*) from Forest
             where Forest.pcode = Park.pcode
             and fsurface > 10
             and (select count (*) from Campground
                  where Campground.fcode = Forest.fcode
                  and fireplace = ``yes'')
                 >
                 (select count (*) from Campground
                  where Campground.fcode = Forest.fcode
                  and not (fireplace = ``yes''))
            )
            >
```

```
(select count (*) from Forest
 where Forest.pcode = Park.pcode
 and not (fsurface > 10)
 or (select count (*) from Campground
       where Campground.fcode = Forest.fcode
       and fireplace = ''yes'')
      <=
      (select count (*) from Campground
       where Campground.fcode = Forest.fcode
       and not (fireplace = ''yes''))
)
```

COOL proves to have also an easy to use natural language structure, in the sense that natural quantifiers are used much as in a natural language. COOL is unique in the field of declarative languages in employing the genitive relations and natural quantifiers, two new concepts of great expressive power.

5.2 Data Model

As mentioned before, COOL's data model is an object-oriented extension of the relational model, the extended Non-First-Normal-Form(NFNF) data model. Instead of normalized relations with atomic-value attributes, as required by the relational model, the NFNF data model allows attributes that contain collections, for example, sets of tuples or relations.

COOL's object data model allows the following as attributes of a non-normal-form relation:

- atomic attributes (e.g.: numbers or strings),
- sets or lists of atomic values (e.g.: list of keywords in a document, or a list of object identifiers to support a relationship),
- structure attributes obtained through aggregation (such as: Address or Date), and
- derived attributes (a function that generates an attribute value from stored attributes' values).

Each object instance in the COOL data model has a unique OID generated by the system. Primary keys can also be used as a meaningful key for the user that does not have access to the OID.

The COOL data model supports the following types of relationships: one-to-many, many-to-many [Bra93a], ISA (one-to-one or generalization or class inheritance) , and recursive many-to-many or one-to-many. It also supports composite objects (aggregation) [Bra93b], aggregation functions (such as: count(), sum(), avg(), and so on), and can be extended with user-defined functions (or methods), user-defined data types (or Abstract Data Types), and to perform encapsulation of data with methods.

5.3 COOL's Basic Concepts

COOL introduces two new concepts to the query language world: **genitive relation** and **natural quantifier**. These concepts give COOL the two unique features mentioned above: the **object-oriented** approach to writing queries and a **natural** language structure.

The OO specification syntax of COOL better fits the majority of cases in which queries deal with aggregation and association of objects. The way in which SQL specifies queries is set-oriented and is best suited to situations where it is natural to deal with whole relations. Unfortunately these situations are rare, for example from Date's well-known query [Dat90]: "retrieve the suppliers that supply all parts listed in the database".

5.3.1 Genitive relation

The genitive relation is a fundamental concept to COOL [Bra94]. It corresponds to the genitive case construct from natural languages. In a query language, this specification technique makes it possible to unambiguously refer to a set of related tuples in a 1:n or n:m relationship.

In order to specify for a certain object, a quantity of related objects that complies with a given condition, COOL uses the following construct:

```
<quantifier> <related_objects> <(condition)>
```

The quantifier symbol could denote any common natural language quantifier: for at least 2%, for the majority, for all, so on. The <related_objects> construct is the **genitive relation** and defines a precise relationship between two object classes, since there could be more than one.

Consider the 1:n relationship between Airline and Aircraft object classes. The reference attribute that defines the relationship, or the **reference list**, is a set of

OIDs of the Aircraft instances that belong to each Airline instance, and is called Aircraft_list.

COOL syntax for the genitive relation used to specify the related object instances of a relationship is: **Airline.Aircraft_list*Aircraft** (1). Thus, the relationship is unambiguously specified by giving the names of the object classes involved and the name of the reference list. When there is only one relationship between two classes the name of the reference list can be omitted. In this case a more natural English-like syntax can be used for the genitive relation: **Airline's Aircraft** (2). The English-like syntax can be used also when there are more than one relationships between Airline and Aircraft so that (1) can be written **Airline's Aircraft_list Aircraft** (3).

Using the relational theory, the syntax above denotes the set of related Aircraft tuples for the Airline tuple. Thus it specifies a relation that can be looked at as the join of the *Aircraft_list* (regarded as a one column relation) and the relation *Aircraft*, using the object identifier as the join field.

Since a genitive relation is a relation, a genitive relation name can also serve as an implicit range variable or tuple variable in COOL, in the same manner as relation names serve as range variables in SQL.

The COOL language syntax and semantics are presented in [Bra93a, Bra93b].

5.3.2 Natural Quantifiers

The use of natural quantifiers in query languages has been given a special attention in a limited number of works. The earliest is [Cha78], where natural quantification is permitted in a limited way in a system called DEDUCE-2 and [Bra78] where natural quantifiers concepts were proposed for a predicate calculus, called EOS. More work on the use of natural quantifiers was done in [Bra83], where a natural quantifier extension of SQL for non-recursive relationships (called SQL/N or SQL/NQ) was proposed, and [Bra88], which introduced a relational algebra operation, called group-select, that used natural quantification of related groups of tuples. In [Bra92c], natural quantifier set theoretic techniques in SQL/NQ are extended for recursive relationships.

The conventional predicate calculus permits only the existential and universal quantifiers. These basic quantifiers are necessary and sufficient in predicate calculus. Although SQL has its roots in predicate calculus, it attempts to 'simplify' the use of quantifiers by replacing the universal quantifier with an equivalent double negation of the existential quantifier. In the writer's opinion, instead of simplifying SQL, this has resulted in users having to deal with highly contrived expressions in which complex structures involving de Morgan's rules need to be used for otherwise quite simple requests. Such requests often involve any of the large number of natural quantifiers, available to the user of natural language. This difficulty, is eliminated by the use of natural quantifiers in a declarative database language.

5.3.3 Genitive relation for 1:n, n:m, and composite 1:n relationships

Consider the Parks database composed only of parks (Park) and forests (Forest) to which we add a new entity, tree species (Tree_Species). Further, let's consider an object-oriented definition of this database which can be graphically represented in an Object-Relationship diagram [Bra92b] as in figure 5.2.

The relationships between the object classes are: a 1:n relationship between Park

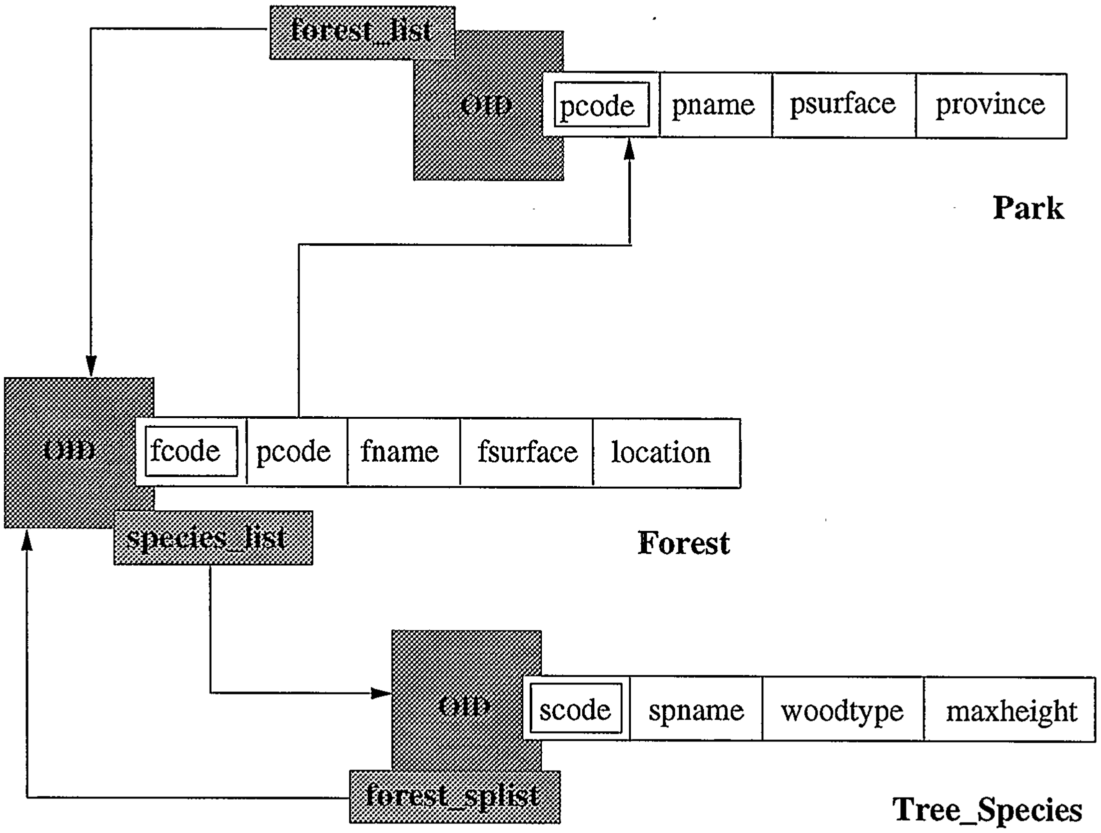

Figure 5.2: Parks Database 2

and Forest (a park contains many forests), and a n:m relationship between Forest and Tree_Species (a forest can have many tree species, and a tree species can be found in many forests).

We can define different kinds of genitive relations with this database.

- **Genitive relation for a one-to-many (1:n) relationship.**

  Query

  Get the park name for each park located in Alberta with at least 4 forests larger than 10 square miles.

  COOL:

```
select pname from Park

where province = ''Alberta''

and for at least 4 Park.forest_list*Forest (fsurface > 10)
```

or

```
select pname from Park

where province = ''Alberta''

and for at least 4 Park's Forest (fsurface > 10)
```

The genitive relation is *Park.forest_list*Forest* and is specified using the list of references: *forest_list*. Alternatively, we can use the alias *Park's Forest* for the genitive relation. Forest_list gives the list of forests larger than 10 square miles for the related parks.

- **Genitive relation for a many to many (n:m) relationship.**

  Query

  Get the forest name for each forest larger than 10 square miles containing a majority of cedar trees.

  COOL:

```
select fname from Forest

where fsurface > 10

and for most Forest.species_list*Tree_Species

                                        (spname = ''cedar'')
```

or

```
select fname from Forest

where fsurface > 10

and for most Forest's Tree_Species (spname = ''cedar'')
```

Many-to-many relationships are symmetrical and a corresponding query for the relationship in the opposite sense is:

Query

Get the name of the tree species which are hardwoods and cannot be found in any forest located in Banff.

COOL:

```
select spname from Tree_Species

where woodtype = ''Hardwood''

and for no Tree_Species.forest_splist*Forest

                                (location = ''Banff'')
```

or

```
select spname from Tree_Species

where woodtype = ''Hardwood''

and for no Tree_Species's Forest (location = ''Banff'')
```

When considering the tree species of a forest we use the genitive relations *Forest.species_list*Tree_Species* or *Forest's Tree_Species*, and when considering the forests that contain a certain tree species, we use the genitive relations *Tree_Species.forest_splist*Forest* or *Tree_Species's Forest*. The n:m relationship used in the above query example is specified in two lists of references, one in Forest called *species_list* and one in Tree_Species called *forest_splist*.

- **Composite 1:n genitive relations**

  In natural language a composite 1:n genitive relation corresponds to: "C objects of the B objects of the A objects". Between A and B and between B and C there are 1:n relationships. Suppose Blist is the set of related B instances for a given A instance and Clist is the set of related C instances for a given B instance. (The general case of a composite 1:n genitive relation may involve 'N' levels of 1:n simple genitive relations, but we restrict it to the most likely case of N = 2.)

  There are two possible ways for the user to construct a COOL expression corresponding to a natural language composite genitive case between the classes A and C (A's Cs, where there is a composite 1:n relationship between A and C):

  - Possibility (1). Specify it in COOL as one composite genitive relation, such as *A.Blist*B.Clist* or *A's B's C*, and
  - Possibility (2). Specify it in COOL using two simple (non composite) 1:n genitive relations, such as *A.Blist*B(B.Clist*C)* or *A's B(B's C)*.

  The two possibilities are both correct only when certain quantifiers are used. In the overall majority of cases, however, the second possibility will give wrong results. To see this, consider the query:

  Name each park in Alberta where all the tree species of the forests (or all the *tree species of all the forests of the park) have the tree maximum height greater than 30 meters.*

In the above query the composite 1:n genitive relation, *the tree species of the forests of the park* used with the quantifier **for all**, can be expressed in the two forms shown above, as follows.

The COOL expression for possibility (1) is:

```
select pname from Park
where province = ''Alberta''
and for all Park's Forest's Tree_Species (maxheight > 30)
```

The genitive relation used in the possibility (1) query is a *single composite genitive relation construct*, Park's Forest's Tree_Species.

The COOL expression for possibility (2) is:

```
select pname from Park
where province = ''Alberta''
and for all Park's Forest
          (for all Forest's Tree_Species (maxheight > 30))
```

In the possibility (2) query, the composite genitive relation is expressed as a combination of two genitive relations, Park's Forest, and Forest's Tree_Species. In this case both possibilities are correct.

But suppose now that instead of the quantifier **for all** the second quantifier specifies a quantity that is neither all nor nonzero, such as **for more than 10**, for example, as in the query:

Name each park where more than 10 of the tree species in its forests have the tree maximum height greater than 30 meters.

This has COOL expression for possibility (1):

```
select pname from Park
where for more than 10 Park's Forest's Tree_Species
                                        (maxheight > 30)
```

And for possibility (2) we might construct:

```
select pname from Park
where for all Park's Forest
          (for more than 10 Forest's Tree_Species
                                       (maxheight > 30))
```

This expression for possibility (2) is clearly wrong. Furthermore, there is no quantifier we can use to replace **for all** to make it correct. And we cannot even do it in an equivalent natural language expression. Therefore, in this case we must use the **composite genitive relation** and **one quantifier**, which is nothing else but, possibility (1).

In the case of a composite genitive relation we are not interested in the quantity of instances from the intermediate classes, that satisfy the condition of the query, all we are interested in is that a join would be possible between the relations involved, Park, Forest, and Tree_Species. Therefore, possibility (1) gives a correct specification for a composite genitive relation for any type of quantifier it uses. It also follows that composite genitive relations in COOL, as with composite genitive case constructs in a natural language, are a necessity.

Possibility (2) works only in the few cases where the quantifier that would be used in the equivalent possibility (1) construct is either the universal quantifier, **for all**, as we saw above, or the existential quantifier, **for at least one**.

5.3.4 Composite Objects

Human thinking is object oriented and a user would prefer to work with composite object instances when dealing with an aggregation-hierarchy database and not with abstract concepts like relations or sets.

The user would prefer to retrieve and store composite object instances, manipulate them with a programming language, have them displayed in a hierarchical format and from a declarative language point of view, and specify the retrieval and update of a composite object instance in terms of the values in that instance.

Consider an aggregation-hierarchy (or composite object) type of database, where the relationships between the entities are mostly one-to-many and involve physical containment or attachment.

For example consider the Provincial parks database with the added object classes Lake and Tree. A park(Park) can contain many forests (Forest) and lakes (Lake). A Forest instance can contain many trees (Tree). Many trees (Tree) can have the same species (Tree_Species), but a tree can belong to only one species. The Object-Relationship diagram for this database is shown in figure 5.3.

The data for a specific park forms a composite object instance involving specific Park data and its contained Forest and Lake instances, with in turn for each Forest instance, the contained Tree instances.

Let's consider the following **composite object** retrieval:

Give full details about Alberta's parks that have at least one forest with most of it's trees spruce trees and with all of its lakes deeper than 4 meters.

The retrieval of the composite object instances is written in COOL, as:

COOL:

```
select *  from Park
          where province = ''Alberta''
          and for at least 1 Park's Forest
                     (for most Forest's Tree
                          (for its Tree's Tree_Species
                                        (spname = ''spruce'')))
          and for all Park's Lake (depth > 4)
       *  from forest_list
       *  from lakes_list
       *  from tree_list
```

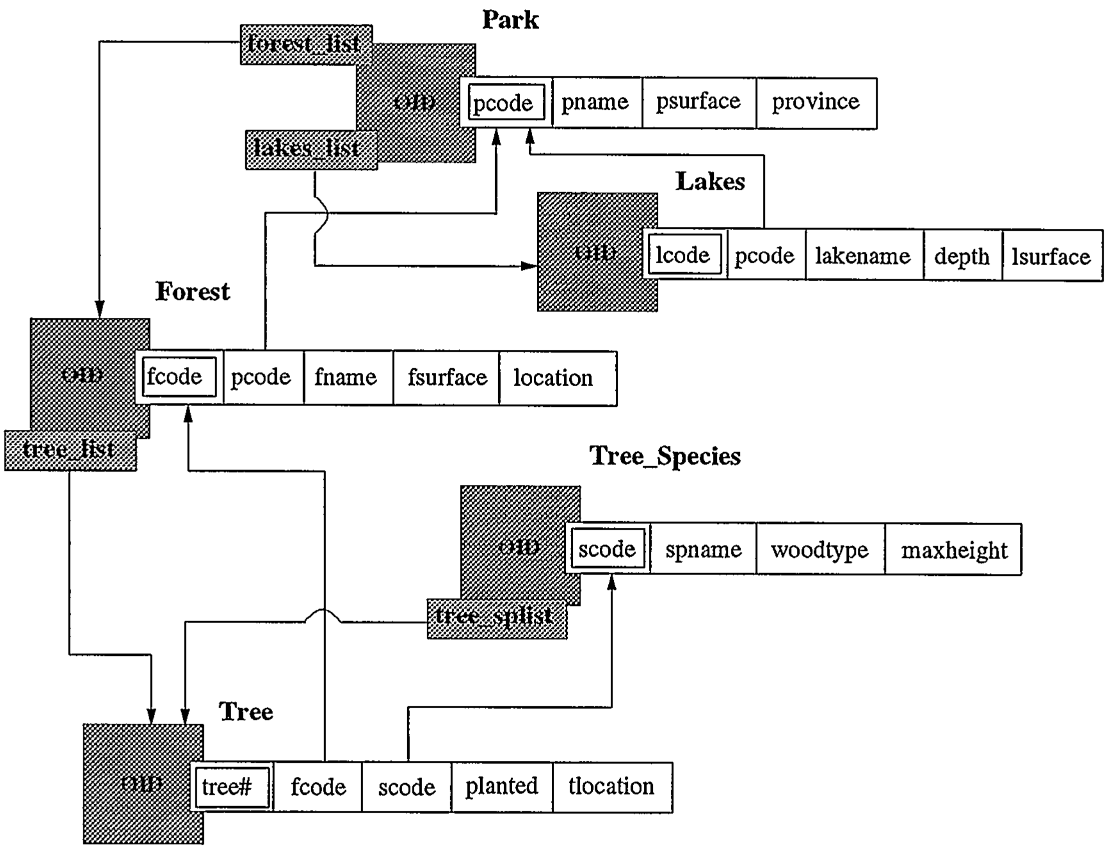

Figure 5.3: Parks Database 3

A composite object (CO) defined as in above can be made into a composite object view, by means of a create composite object [view] command, that can also name the CO. Also the CO can be concentrated (certain subobjects are omitted from a composite object using supplementary conditions). The concentrated CO can be put in a view as well. Language constructs for retrieving, concentrating and creating views of composite objects are presented in [Bra93b].

5.3.5 Functions

The functions in COOL can be used to compute non stored attributes from other attributes specified in the database definition. An example would be the function Area(), for use in calculating the area of a spherical object when the radius is a stored attribute. Let's take the following example: A Sphere object class is defined as:

COOL:

```
create obj cls Sphere
      ( S# char(4),
        x  int,
        y int,
        z int,
        r int
        area() int function,
      )
```

where x, y, z are the coordinates of the center and r is the radius. Area() and Volume() functions can be defined as non stored attributes on stored attributes of the Sphere. If the stored attributes are all private and only function attributes are public, a partial encapsulation is accomplished.

COOL can use special purpose user-written functions defined as attributes as well, such as the function Overlap(). Overlap() can be defined for use with Sphere objects. It would be true if a specified sphere physically overlaps the Sphere instance for which a condition written in COOL, holds.

COOL:

```
select s# from Sphere
where overlap(select s# from Sphere
                 where x = 8 and y = 10 and z = 15 and r = 20)
```

or

```
select s# from Sphere
where overlap(s2)
```

would retrieve those Sphere instances that overlapped the specific Sphere specified in the COOL expression used as a function parameter.

5.4 Extended Relational Algebra operations

The reduction of natural quantifier expressions defined in COOL requires special relational algebra operations. An Extended Relational Algebra (ERA) has been developed for COOL and was presented in [Bra94]. It consists of conventional relational algebra operations (select, project, join, intersection, union) and three unconventional operations: group-select, subgroup-select and possibility join.

5.4.1 Group-select operation

This operation is found to be useful for efficiently reducing a natural quantifier expression (i.e. an expression with a quantifier in it, e.g.: for at most 10 Park's Forest). It adds counting facilities to conventional algebra. Group-select operation was first introduced in [Bra88].

The group-select operation has the syntax:

```
RR = group-select( R (q S (c)))
```

The group-select of relation R with the foreign key called S is a relation RR that contains all groups of tuples with the same attribute value of S from R, provided for each of them a quantity q of tuples satisfies the condition c.

As an example of a query with the quantifier **for all**, on the database from figure 5.3, consider:

Query

Get the name of each park with all of its forests having the area greater than 15 square miles.

COOL:

```
select pname from Park
where for all Park's Forest (fsurface > 15)
```

ERA:

```
R1 = group-select (Forest (for all pcode (fsurface > 15)))
R2 = Park(pcode) join R1(pcode)
R3 = project (R2 (pname))
```

Relation R1 will contain all the groups of tuples from Forest such that, within each group of Forest tuples with the same **pcode** value, all the tuples of the group satisfy the condition that **fsurface** is greater than 15. In order to get the information we need about parks, relation R1 will be joined with relation Park, on the attribute **pcode**, giving the result R2. Furthermore, the desired result will be obtained by projecting R2 on the field **pname**.

The general form of a retrieval that involves a group-select operation is:

Retrieve each RP tuple for which a RP condition (compound condition involving RP attributes: Pa1,..., Pan) holds and for which a specific quantity of related RC1 tuples obey the RC1-condition (a compound condition involving RC1 attributes: C1a1,...,C1an).

RP and RC1 belong to the following hierarchy of relations:

```
RP (P, Pa1,...,Pan, RC1_list,..., RCn_list)              (5.1)
RC1 (P, C1, C1a1,...C1an, RC11_list,...,RC1n_list)      (5.2)
```

where there is a 1:n relationship between RP and RC1, supported by the reference list RC1_list.

The general COOL expression and ERA routine are:

COOL:

```
    select * from RP
    where (RP_condition)
    and quantifier RP.RC1_list*RC1 (RC1_condition)
```

ERA:

```
    R0 = select (RP (RP_condition))
    R1 = group-select (RC1 (quantifier P (RC1_condition)))
    R2 = R0(P) join R1(P)
    R3 = project (R2 (*))
```

5.4.2 Possibility Join

As shown in the previous section group-select operation solves the problem of reduction of natural quantifier expressions with a single level of nesting. Quantifier expressions with more than one level of nesting can be solved by the possibility join operation combined with group select. A nested natural quantifier expression is an expression with a natural quantifier, in which a further expression with a natural quantifier is embedded. An example later will make this clear.

The possibility join (pjoin) operation applied to relations A and B, is written as:

```
RR = A(m) pjoin(p) B(m)
```

In the result relation RR there will be placed every tuple from A plus an additional attribute **p**, called the **join possibility attribute**. The value of **p** in an RR tuple is *true* if there is a tuple in B with the same **m** value, otherwise it is *false*. Thus the **p** attribute in an RR tuple indicates whether or not it is possible to join a A tuple with a B tuple. The pjoin operation resembles the outerjoin operation. The only difference is the supplementary field that is added to the result.

Consider a retrieval with 2 levels of natural quantifier expressions:

Find the parks located in British Columbia, where most forests are larger 15 square miles and have all the trees planted before 1989.

COOL:

```
select pname from Park
where location = ''British Columbia''
and for most Park.forest_list*Forest (fsurface > 15
    and for all Forest.tree_list*Tree (planted < 01011989))
```

ERA:

```
R1 = group-select (Tree (for all fcode (planted < 01011989)))
R2 = Forest (fcode) pjoin(p) R1 (fcode)
R3 = select (Park (location = ''British Columbia''))
R4 = group-select (R2 (for most pcode (fsurface > 5 and p)))
R5 = R3(pcode) join R4(pcode)
R6 = project (R5 (pname))
```

R1 will contain all the groups of tuples from Tree that have the value of attribute **fcode** matched with the value of **fcode** from Forest, and within each group of Tree tuples with the same **fcode** value, all the tuples satisfy the condition that the date of plantation, **planted** is less than the Jan 1st 1989.

The **pjoin** operation has placed in R2 all the Forest tuples joined with the R1 tuples on the field **fcode**, and has concatenated each of the resulting tuples with a an attribute **p**. The value of **p**, corresponding to a Forest tuple, is *true* if there is a tuple in R1 with the same **fcode** value, otherwise it is *false*. However, the only case when **p** = **false** is taken into account is the case of **for majority** or **for most** quantifier. R2 is further processed by a group-select operation that will select all the groups of tuples from R2 that have the value of **pcode** matched with the value of **pcode** from Park and within each group most of the tuples satisfy the condition **fsurface** $>$ 5 and have a *true* value for **p**, that is the tuples could be joined with the tuples in R1. The result of group-select, R4 is further joined with selected tuples from Park, for which *location = "British Columbia"*. The final result R6 is obtained by projecting the result of the previous join on the field **pname**.

In the general case of possibility join, we have added a third level to the hierarchy in (5.1) and (5.2), such as:

```
RC11 (C1, C11, C11a1,..., C11an, RC111\_list,..., RC11n\_list)  (5.3)
```

so that the general COOL expression for this three level hierarchy is:

```
select RP_attribute_list from RP
where (RP-condition)
and  C1quantifier RP.RC1_list*RC1 (RC1-condition
          and C11quantifier RC1.RC11_list*RC11 (RC11-condition))
```

The expressive power of COOL becomes clearer when one reflects that the above expression can involve nested quantifier expressions with many different types of quantifiers. Writing the equivalent in SQL would involve a large set of quite complex SQL expressions, often requiring the use of De Morgan rules with nested expressions.

The equivalent ERA routine is:

```
R1 = group-select (RC11 (C11quantifier C1 (RC11-condition)))
R2 = RC1(C1) pjoin(p) R1(C1)
R3 = group-select (R2 (C1quantifier P (RC1-condition and p)))
R4 = select (RP (RP-condition))
R5 = R4(P) join R3(P)
R6 = project (R5 (RP_attribute_list))
```

5.4.3 Subgroup-select

If in the group-select instead of the requirement that the quantity **q** of the tuples of R with the same foreign key value A satisfy a condition **c**, we have the requirement that a quantity **q** of selected tuples of R with the same A value [that satisfy a condition c1] also satisfy the condition **c2**, then a **subgroup-select** operation can be defined.

A subgroup-select operation is defined as follows:

RR = subgroup-select (R (q (A(c1)) (c2))),

This specifies that RR will contain the sets of tuples of R with the same value of attribute A, such that: for each such set of R tuples, a subset of tuples for which c1 holds is considered, and if the quantity q of this subset satisfies c2, the original R set is placed in RR, otherwise it is not.

For example, compare the two retrievals:

- **Retrieval type A**

  Get the names of parks located in Washington state where most of the lakes are over 6 meters in depth and are larger than 5 square miles.

  COOL:

```
select pname from Park
where location = ''Washington''
and for most Park.lakes_list*Lake (depth > 6
                                   and lsurface > 5)
```

- **Retrieval type B**

  Get the names of the parks with most of the over-6-meter-deep lakes are larger than 5 square miles.

  COOL:

```
        select pname from Park
        where location = ''Washington''
        and for most Park.lakes_list*(Lake (depth > 6))
                                              (lsurface > 5)
```

In example B we need a specific quantity, **for most** of not just the lakes of a park, but **for most** of the lakes over 6 meters deep, that is, for most of a specific subset of the lakes of the park.

The type of the quantifier is very important in retrievals of type B. With an existential type of quantifiers, for example, **at least 2**, **at least 4**, etc., retrievals of type A and B above are equivalent. The retrievals of type A and B become fundamentally different when a universal type quantifier is involved. A universal type quantifier, such as **for all, for all but 2, for one and all, for most** [**of all**], refers to all of the tuples from the group being evaluated, and of course we do not know in advance what constitutes this group. That is why we need to select a group of tuples by applying condition **c1** first to R tuples with the same A value and then narrow the selection by applying the condition **c2** and the quantifier to the group of tuples initially selected.

ERA routine for query B is:

```
R1 = select (Park (location = ''Washington''))
R2 = subgroup-select
     (Lake (for most (pcode (depth > 6)) (lsurface > 5)))
R3 = R1(pcode) join R2(pcode)
R4 = project (R3 (pname))
```

A general retrieval of type B, is:

Retrieve each RP tuple for which RP-condition (compound condition involving attributes Pa1,..., Pan) holds and for which a specific quantity of all of the related RC1 tuples for which RC1-condition1 (condition involving RC1's attributes) holds , obeys the RC1-condition2 (condition involving attributes C1a1,...,C1an).

The equivalent COOL expression is:

```
select * from RP
where RP-condition
and quantifier RP.RC1_list*(RC1 (RC1(RC1-condition1))
                (RC1-condition2)
```

and reduces to: **ERA:**

```
R1 = select (RP (RP-condition))
R2 = subgroup-select
    (RC1 (quantifier (P (RC1-condition1)) (RC1-condition2)))
R3 = R1(P) join R2(P)
R4 = project (R3 (*))
```

Chapter 6

Implementation

6.1 Overview and general issues

For the prototype implementation of COOL there were essentially two choices:

1. Implementing a completely new database system based on the data model, and

2. Building a front-end system on top of an existing system.

Since building a completely new DBMS with all the necessary components is a complex task that requires large time resources, the second option of a front-end to an existing relational database system (e.g.: Sybase, Oracle), was adopted.

There are two basic methods of reducing COOL expressions:

1. The reduction of COOL directly to SQL, and

2. The reduction of COOL expressions to Extended Relational Algebra (ERA) routines.

The first approach of translating COOL expressions directly into collections of SQL expressions would result in a system which would not be useful at some later time if a complete object-relational DBMS were to be built.

An approach that would offer portability and flexibility for future development of a full DBMS is the second approach that requires the reduction of COOL expressions to ERA routines. The ERA routines could be converted into SQL or into programming language routines (such as C or C++). But converting ERA routines to C/C++ routines is equivalent to the building of a complete DBMS, an option that we have already rejected. Thus, the best choice of the prototype implementation of a front-end system is the translation of COOL to ERA routines and then to SQL expressions. Since SQL is a standard, the prototype front end can run on any relational system. However, the current prototype version of COOL has been implemented as a front end for the Sybase storage manager. It was also written in C.

6.2 Comparison with other prototype implementations

Other developers have taken a similar approach to prototype implementations. An example is OSCAR (Object management System Clausthal, Approach: Relational), described in [HFWC91]. OSCAR's data model, called EXTREM (EXTended RElational Model) is a semantic data model (a subset of IFO semantic model) equivalent to an extended nested relational model whose algebra can be mapped to conventional relational algebra. A prototype of OSCAR was implemented on top of IRIS relational storage manager.

There are also developers who have reduced language expressions directly to SQL. For example, in [KR90] is described an object-oriented SQL front-end (OOSQL) for the IBM DB2 relational database. The OOSQL commands are translated by the OOSQL interpreter into DB2 SQL statements.

The prototype front-end for COOL was designed as a three-layered application.

1. The first layer is a shell interface that accepts the command statements for

COOL source file translation and outputs the results, or error messages, generated by the database system.

2. The second layer is responsible for the first step of COOL translation. COOL statements (filename.cool) are reduced to Extended Relational Algebra routines and data structures are prepared for the following layer.

3. The third layer handles the translation of ERA routines (filename.era). ERA routines are translated to a set of SQL statements (filename.sql). Catalog management is also performed in this layer.

6.3 Implementation Design of COOL's object model

The object-oriented data model of COOL follows the object-relational approach [SRL+90], that is, it includes OO extensions to the relational model.

6.3.1 Implementation levels

The implementation employs three levels of abstraction. Beginning with the highest, we have:

1. **Conceptual level.** This level uses a structurally object-oriented or semantic database model, such as the Entity Relationship model extended by ISA relationships and complex attributes.

2. **Extended relational or object-oriented level.** This level uses an extension of a non-first-normal-form relational model and is an equivalent representation of the E/R concepts of the previous level.

3. **Conventional relational or implementation level.** This level is used as a basis for the prototype implementation and is an equivalent representation using the conventional relational model of the previous two levels.

A similar approach was used in [HFWC91] for the implementation of an object-oriented database system with an object algebra based on a modified nested relational algebra.

An **aircraft maintenance database**, was chosen to be used for example queries throughout of this chapter. The schema of this database, which at the conceptual level uses a data definition language proposed in [SS91, Cat91, Bra93a] for an object-oriented data model obtained by extending the relational model, is shown in the Appendix B.

The aircraft maintenance database has a network database structure containing one-to-many (1:n), many-to-many (n:m), and recursive many-to-many relationships.

How COOL's OO schema above can be mapped into the conventional relational schema is shown by drawing the diagrams corresponding to the three levels of abstraction. These three levels are illustrated in figure 6.1 and in figure 6.2 (the conceptual level), in figure 6.3 (the object-oriented level), and in figure 6.4 (the implementation level).

At the conceptual level (figure 6.1 and figure 6.2) we distinguish between entity sets or object types, relationship types and attribute values of these entity types belonging to attribute types. Every entity type has underlying attribute types. Every relationship type can also have underlying attribute types (e.g.: relationship 'owns' in figure 6.1 and in figure 6.2). Attribute types can be simple, such as strings and integers (e.g.: description), or constructed via constructors such as set (e.g.: Aircraft_list in figure 6.1). For each entity set we can define a set of keys consisting of attribute or relationship type. As a special case, relationships can be restricted to be 1:1 or 1:n by these keys. In figure 6.1 and figure 6.2, rectangles represent entity types. Empty ovals represent simple attributes. Bold ovals represent constructed attributes. Diamonds represent relationships.

At the extended-relational level or the OO level (figure 6.3), the conceptual level is mapped into the NFNF relational level, as follows. Each object type describes a set of object instances with identical structure. An object type maps into a NFNF relation and implicitly each object instance maps into a NFNF tuple. For each object instance, a system generated unique object identifier (OID), will be assigned. Relationships are mapped into reference attributes (e.g.: Job_list is a set of Jobs).

The extended-relational level maps into the conventional relational level or the implementation level (figure 6.4), as follows. Each NFNF relation maps into a conventional relation and each NFNF tuple maps into a conventional tuple. For mapping the reference attributes (sets or lists) that describe the relationships between the object classes, we have different approaches. Some of them were discussed in Chapter 2.

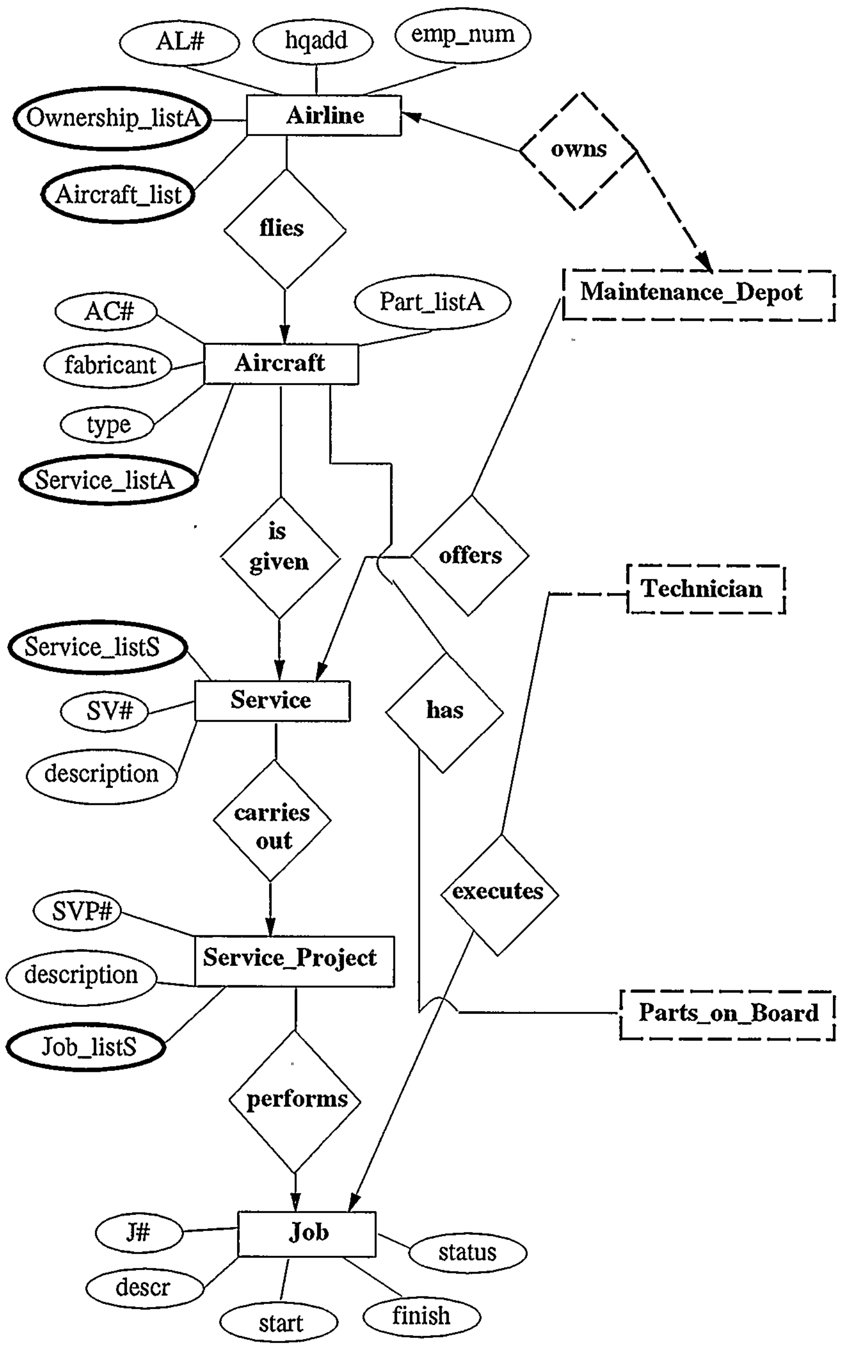

Figure 6.1: The Entity Relationship diagram for the conceptual level of the aircraft maintenance database, part1

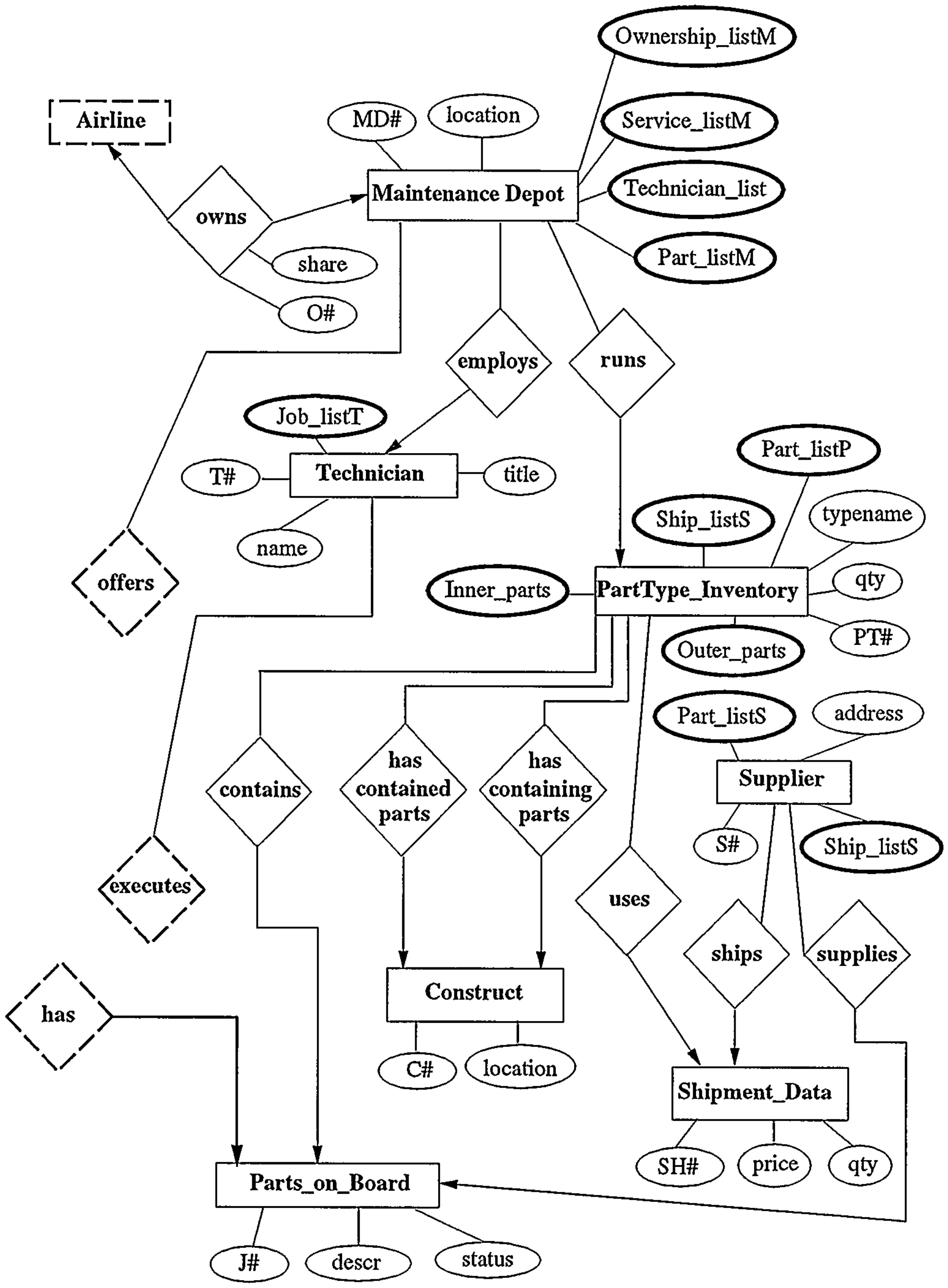

Figure 6.2: The Entity Relationship diagram for the conceptual level of the aircraft maintenance database, part2

With reference to figure 6.3 the following diagrammatic conventions are used:

- Bold rectangle —> object class.
- Normal rectangle —> reference attribute.
- Dashed rectangle —> foreign key reference attribute.
- Hatched rectangle —> primary key.
- Shaded square —> object identifier.

With reference to figure 6.4 the following diagrammatic conventions apply:

- Bold rectangle —> relation.
- Normal rectangle —> foreign key.
- Shaded rectangle —> primary key.

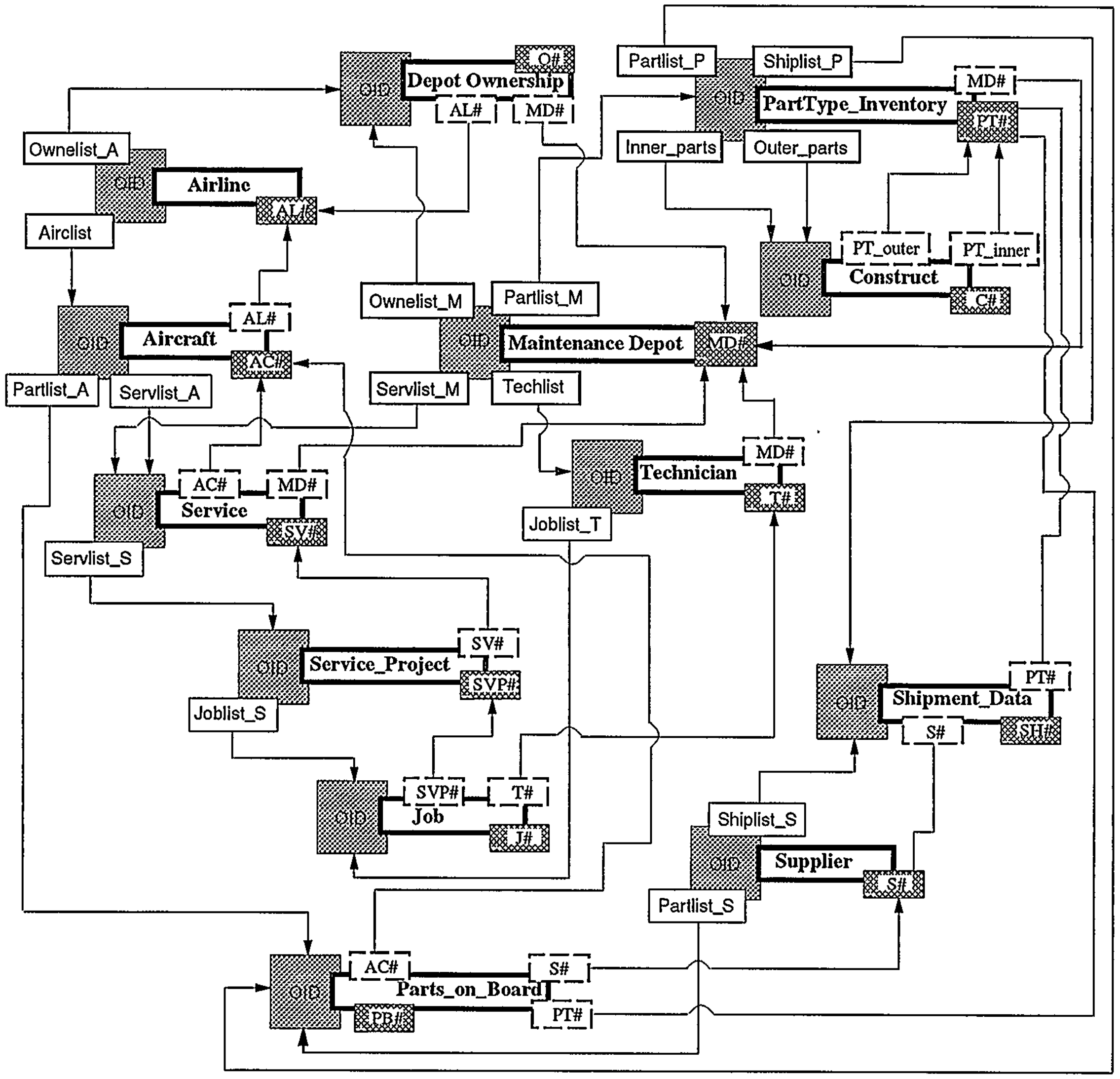

Figure 6.3: The Object Relationship diagram for the object level of the aircraft maintenance database

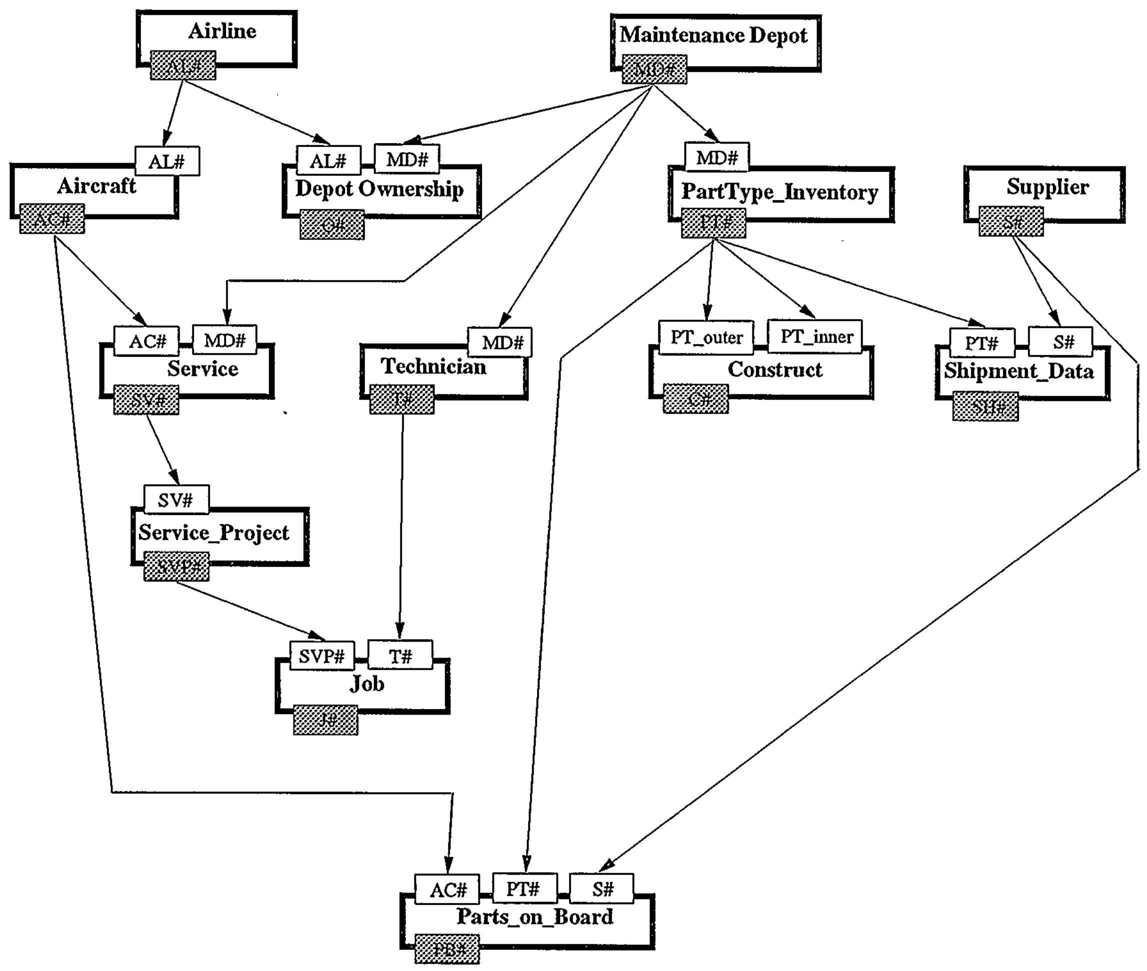

Figure 6.4: The relational diagram for the implementation level of the aircraft maintenance database

6.3.2 Mapping relationships into the relational model

Basically the aircraft maintenance database can be regarded as a collection of relations that participate in 1:n relationships. Many-to-many and the recursive many-to-many relationships in the database are supported using 1:n relationships.

The 1:n relationship between object classes can be implemented in the relational model either (1) as a relation or (2) using the classical concepts of primary key and foreign key, where the foreign key is stored in the child relation of the 1:n relationship. For the second approach I had the choice of using either OIDs or the primary keys of the related classes, as foreign keys.

In the current implementation I have combined the two approaches. If the database has primary keys and foreign keys at the conceptual level, these keys should be kept in the schema, because this is the way in which the user can define relationships. At the internal level, though, an OID could replace the primary key, and implicitly the foreign key values.

However, when a new object instance is inserted into the database any matching between instances of related object classes is accomplished by mean of the values of primary and foreign keys, because the user does not have access to the OID. For this reason I decided to use primary key values and not OIDs as foreign keys.

A 1:n relationship was also implemented as a table (relation) containing matching OIDs of the parent object instances and of the children object instances. The 1:n relationship tables are created dynamically and maintained by the system. The user does not have access to them. Further details about these tables will be given in the COOL object catalog section.

Thus, I use the foreign key concept because is closer to the user's view of the database and I use the table to implement relationships because it makes it easier to upgrade the system to support direct n:m relationships. A many-to-many (n:m) relationship between relations A and B can usually be expressed as a join of two 1:n relationships between each A and B and a third relation Z.

If the n:m relationship is implemented as two 1:n relationships, such as A 1:n Z and B 1:n Z, then the direct link A n:m B can be obtained by joining the tables that implement the relationships A to Z and B to Z. Tables A and B must contain the primary key or OIDs of Z. Thus, an n:m relationship can be implemented either using two relationship tables, where each table implements a 1:n relationship, or it can be implemented as a single table.

For n:m relationships, the single table solution will require less computation time, but will need more storage space; instead of a join between two tables to get the related tuples of the n:m relationship, with a single table solution the join is already there in the relationship table.

6.4 Design and implementation of the Data Definition Language (DDL)

In relational databases, a DDL is used to specify the database schema. A conceptual database schema specifies a set of entity types. For each entity type, a set of attributes with their domains, as well as integrity constraints on the domains, are also specified. The DDL for COOL is specified in the same way.

The DDL for COOL contains a CREATE OBJECT CLASS statement. For the

CREATE statement syntax two implementation approaches were considered.

1. A class and its relationships must be declared in the same CREATE statement,

2. A class and its relationships must be declared separately.

In the current implementation the first approach was chosen, so that a class and its relationships are specified in one CREATE statement. This option was chosen because insert operation involving child instances needs information about the parent of such children and this information is provided by the super key (**super key** has the same meaning as **foreign key**). The second approach is analyzed in Chapter 8.

The syntax for the CREATE STATEMENT, in BNF format, is as follows:

```
<create_objectcls> ::=
               CREATE OBJ[ECT] CL[AS]S <objectcls>
                        '('<objectcls_element_commalist> ')'
<objectcls_element_commalist> ::=
               <objectcls_element>
             | <objectcls_element_commalist> ',' <objectcls_element>
<objectcls_element> ::=
               <attribute_def>
             | <objectcls_key_def>
<attribute_def> ::=
               <attribute> <attribute_type>
<objectcls_key_def> ::=
               CANDIDATE KEY '(' <attribute_commalist> ')'
             | PRIMARY KEY <attribute>
             |  SUPER KEY '(' <relationship_field_commalist> ')'
<relationship_field_commalist> ::=
             <relationship_field>
           | <relationship_field_commalist> ',' <relationship_field>
<relationship_field> ::=
                 <attribute> '(' <parent_objectcls> ')'
                          [':' <reference_attribute> ')']
<attribute_commalist> ::=
                 <attribute>
               | <attribute_commalist> ',' <attribute>
```

The ISA relationship was not implemented. The relational algebra for implementing an ISA relationship is discussed in Chapter 8.

The declaration of an 1:1 or ISA relationship can be done inside the CREATE statement, in the < *objectcls_key_def* > definition, by adding :

```
SUPERCLASS KEY '(' <relationship_field_commalist> ')'
```

Some examples of CREATE statements for the aircraft database are:

```
create object class Airline
( AL#    CHAR(2),
  hqadd    char(30),
  emp_num  int,
  primary key AL#
);
create obj cl Service
```

```
(S#     char(4),
 AC#     char(4),
 MD#     char(4),
 description char(80),
 primary key S#,
 super key (AC# (Aircraft), MD# (Maintenance_Depot))
);
create obj cl Construct
(code#  char(4),
 PT_outer char(4),
 PT_inner char(4),
 location char(10),
 primary key code#,
 super key (PT_outer (PartType_Inventory:containing_parts),
            PT_inner (PartType_Inventory:contained_parts))
);
```

Candidate keys (see CANDIDATE KEY in the CREATE STATEMENT above) are allowed for user convenience, but have no other system significance. Any unique attribute that can serve as a primary key is a candidate key.

Reference attributes names are optional in the syntax definition (see the AC# and MD# in the definition of the object class Service and also figure 6.3). If they are not supplied by the user, the reference attribute names will be generated by the system, as unique combinations between the names of the connected classes. When there is more than one relationship defined by reference attributes between the same two object classes (see PT_outer and PT_inner in the definition of class Construct and also figure 6.3), it is better if the user gives meaningful names for the reference attributes, instead of leaving this task to the system. The system uses a combination of the names of the parent and child object classes to generate a reference attribute name.

The attribute types implemented are integers and strings of characters. The primary key is a single field primary key. The primary key gives the user useful information. It can be replaced by the OID, but then we gain efficiency (we eliminate an attribute) to the detriment of information. A composite primary key would not change the translation algorithm. However, an increase in complexity would appear in the join operations on the primary key 'field' (in ERA expressions) that translate to multiple SQL joins. Composite primary keys are avoided by defining a new attribute as primary key. Another way to avoid composite keys is by declaring OIDs as primary keys.

6.4.1 COOL's Object Catalog

The term COOL's **Object Catalog** is used to refer to all the Sybase tables that need to be generated by the system, in order to built COOL's object-oriented constructs, such as genitive relations and object identifiers.

COOL's Catalog tables are:

- A. **The table *cool_objects*.** This table is used for generating unique OIDs for object instances, and contains the following information:

- *class_name* defines the object class name, and
- *next_id#* is the next value to be assigned to a new object instance in that class.

SQL:

```
create table cool_objects
  (class_name char(30),
   next_id# int
  )
```

The OID generated is unique in each class. If the *class_name* were not associated with the OID, then the uniqness of an OID for the whole database could be achieved by appending a class code to the OID. This is particularly useful when defining complex objects. If instead, the system had been designed to generate a unique object instance OID for the whole database without keeping track of the class an object instance belongs to, this would have been a poor design because of the loss of the link between object instance and object class. The algorithm for generation OIDs is the simplest possible to assure uniqness for a code: just allocate a positive integer to the current OID, and then increment it by one.

- B. **The table *cool_keys*.** This table keeps track of the keys defined in a CREATE statement , and contains the following information:
  - *class_name* defines the object class name,

- *key_name* defines the name of the key, and
- *key_type* is a codification for the type of the key.

SQL:

```
create table cool_keys
  (class_name char(30),
   key_name char(30),
   key_type char(20)
  )
```

The table *cool_keys* is necessary because the keys defined in a CREATE statement have different meanings from the keys in Sybase.

In the field *key_type* are stored codes for primary keys, reference lists, candidate keys and super keys. The code of a super key specifies also the name of the reference list that corresponds to the superkey (e.g. *class_name = "Aircraft"*, *key_name = "AL#"*, *key_type = "SK Airclist_Air"*, where *Airclist_Air* is the name of the reference_list that implements the 1:n relationship between the parent *Airline* and the child *Aircraft*.

Table *cool_keys* is updated every time a new object class is created or dropped and table *cool_objects* is updated every time an object instance is inserted.

- C. **Tables that implement 1:n relationships (or genitive relations)**. The name of each relationship table is generated from the name of the parent class, child class and the reference_attribute (i.e. the name of the relationship). The relationship tables are created or updated whenever child instances are inserted or deleted in the database.

A relationship table has two attributes:

1. The OID of the parent object and
2. The OID of the related children object.

The prototype system maintains the object catalog tables using the Sybase interface DB/Library to the SQL manager [Syb91a].

6.5 Translation of COOL

The translation of COOL DML **select** expressions is carried out in two steps:

1. Translation of COOL to ERA, and
2. Translation of ERA to SQL.

Since COOL is a declarative database language, it has a DDL and a DML. Among the statements of the DML, only the select statement is translated to ERA routines. All the other manipulative statements (delete, update, and insert) and the create statement are translated directly to SQL.

The implemented COOL's grammar is in Appendix A. In writing the translators two compiler-construction tools (Unix packages) were used, namely *lex* and *yacc* [MB90]. Besides their help for automatic design of specific compiler components, these tools make the future development of the language much easier (instead of modifying a program, one needs to modify only a short specification). Basically a compiler is a program that reads a text file that contains program code written in a source language and translates it into an equivalent program written in a target language [ASU86]. The basic phases of a compiler are (1) lexical analysis, (2) syntax analysis or parsing and (3) code generation.

The first phase consists of a lexical analyzer whose job is to scan the source file and match sequences of characters that identify tokens. Tokens are essentially sequences of characters having a collective meaning. Lex reads a specification file and generates a C routine that performs lexical analysis based on a finite automata. The lex specification contains the regular expressions for pattern matching and the actions associated with them. The actions in a lex specification consist of C language statements that return the token number and value, if any.

In the second phase of a compiler, a parser reads tokens of the source program and assembles them into grammatical phrases. Further on, the grammatical phrases are used by the third phase of the compiler to synthesize the output. Yacc reads a specification file that codifies the grammar of the source language and generates a C parsing routine. The yacc specification contains the source language grammar rules and the corresponding semantic actions.

When the parsing routine detects a sequence of tokens that corresponds to a grammar rule, an associated action is executed. The actions in yacc are one or more C statements that make use of the values of tokens either to generate output or to pass the value to other routines in the program. Many programming languages have a recursive structure that can be defined by context-free grammars. The most common notation used to describe a context-free grammar is known as BNF (Backus-Naur Form). The grammar rules in the yacc specification closely follows BNF. COOL's grammar is also a context-free grammar [Sal73].

6.5.1 First step. Translation of COOL expressions to ERA or/and SQL

This step was implemented as a separate compiler that reads a file of COOL expressions and translates it into a file of ERA routines in the case of COOL 'select' expressions, or into a file of SQL expressions for other COOL expressions. The lex file for this step will contain the regular expressions for COOL's tokens and the associated actions, and the yacc file will contain the COOL's grammar production rules and the corresponding actions for code generation in ERA or SQL.

6.5.2 Reduction of COOL Queries to Extended Relational Algebra (ERA)

In generating ERA routines, basic rules in query processing optimization have been applied everywhere. Some of the rules are:

- perform selection and projection first,
- replace a join by a semijoin, and
- reorder operations to reduce intermediate relation size.

I have also applied my own rule: *keep only what is needed for the next step.*

A COOL query (or select statement) has an SQL-like structure, SELECT / FROM / WHERE. In COOL, selection specified in the SELECT field is carried out only on attributes belonging to the class specified in the FROM field. Only one class is allowed in the FROM field. The WHERE expression is the most complex one, because it specifies nested quantifier expressions involving the relationships (1:n, n:n, recursive many-to-many) associated with desired class. The relationships are embodied in COOL by the construct called the genitive-relation, which was introduced earlier.

The translation process of the COOL queries was built using a top-down approach. Beginning with simple queries, basic translation rules were developed. Queries are basically one level (simple) or multiple nested levels (complex). For the nested queries with multiple levels an induction approach was used. This involves generating the algebra routines for a one and two levels of nested queries, and then generalizing the algorithm for n levels. The following translation rules were applied: (1) translation starts with the **where clause**, if the **where clause** is not empty and the query is one level query, (2) if the query has multiple nested levels, the translation starts with the deepest level. Since a yacc grammar was used, the derivation rules establish the order in which grammar rules are reduced.

In the following sections different types of genitive relations involved in nested COOL query expressions will be analyzed in the context of reduction of these expressions to ERA routines. The genitive relations analyzed are: simple parent-to-child, composite parent-to-child, simple child-to-parent, composite child-to-parent. For each of the genitive relation types above we analyze two cases: one level queries and multiple levels (or nested) queries.

6.5.3 Parent-to-Child Genitive Relations

1. Simple Parent-Child genitive relation corresponding to a 1:n relationship.

1a) One level query.

When a parent and its children are involved in a query we will deal with only one level of quantifier expressions (or expressions that define a quantity of related instances).

For example, suppose we take the retrieval:

Get full details of each Airline headquartered in NY that uses only Boeing aircraft.

The COOL expression is:

```
select * from Airline
where hqadd = ''NY''
and for all Airline's Aircraft (fabricant = ''Boeing'');
```

The above query has the following equivalent ERA routine:

```
R0 = group_select
           (Aircraft (for all AL# (fabricant = "Boeing")))
R1 = select ( Airline ( hqadd = "NY"))
R2 = project ( R1 ( AL#))
R3 = R0 (AL#) join Airline (AL#)
R4 = project ( R3 ( AL#))
R5 = R4 intersect(AL#) R2
R6 = R5 (AL#) join Airline (AL#)
R7 = project ( R6 (*))
```

If instead of ***and** for all Airline's Aircraft* we have had ***or** for all...* , R5 from the ERA routine will become R5 = R4 *union* R2. Thus, as a general rule, in COOL expression, *ands* will generate *intersections* and *ors* will generate *unions* in the corresponding ERA routine.

Suppose now that we have one parent with two children involved in the query, and consider the retrieval:

Give the airline codes for the airlines with headquarters in San Diego that fly mostly MacDonald Douglas aircraft and own outright at least 2 maintenance depots.

The COOL expression and the corresponding algebra will be as follows:

COOL:

```
select AL# from Airline
where hqadd = ''San Diego''
and for most Airline's Aircraft (fabricant = ''MacDonald Douglas'')
and for at least 2 Airline's Depot_Ownership (share = 100);
```

ERA:

```
R0 = group_select
     (Aircraft (for most AL# (fabricant = "MacDonald Douglas")))
R1 = group_select
     (Depot_Ownership (for at least 2 AL# (share = 100)))
R2 = R0 intersect(AL#) R1
R3 = select ( Airline ( hqadd = "San Diego"))
R4 = project ( R3 ( AL#))
R5 = R2 (AL#) join Airline (AL#)
R6 = project ( R5 ( AL#))
R7 = R6 intersect(AL#) R4
R8 = R7 (AL#) join Airline (AL#)
R9 = project ( R8 (AL#))
```

In the general case, retrievals involving a parent with n children and one level of nested quantified expressions can be formalized using an extended-relational model as a 1:n hierarchy of relations as follows.

In general, suppose we have a parent relation RP with the children relations RC1, RC2,..., RCn. Suppose the relation RP has the attributes: P (primary key), Pa1, Pa2,..., Pan, RC1_list, RC2_list,..., RCk_list,..., RCn_list. RCk_list is a set of tuples from relation child RCk, describing the 1:n relationship between RP and RCk.

Suppose also that the child relations have the following description:

RC1 (P, C1 (primary key), C1a1, C1a2,..., C1an, RC11_list, RC12_list, ..., RC1_n_list);

RC2 (P, C2 (primary key), C2a1, C2a2,..., C2an, RC21_list, RC22_list, ..., RC2_n_list);

The general retrieval for one level of nested quantified expressions is:

Retrieve each RP tuple for which RP - condition (compound condition involving fields: Pa1, Pa2,..., Pan) holds and for which a specific quantity of related RC1 tuples obey the RC1 - condition (compound condition involving C1a1, C1a2,..., C1an) and a specific quantity of related RC2 tuples obey the RC2 - condition and so on for 'n' children.

The general COOL query for above is:

```
select * from RP
where (RP - condition)
and/or quantifier1 RP's RC1 (RC1 - condition)
and/or quantifier2 RP's RC2 (RC2 - condition)
```

```
...
and/or quantifierN RP's RCn (RCn - condition)
```

The ERA routine (referred to later in Section 6.5.5 as ERA-Gen1) for the general retrieval involving one level of nested quantified expressions will be:

```
R1 = group-select
       (RC1 (quantifier1 foreignkey (RC1 - condition)))
R2 = group-select
       (RC2 (quantifier2 foreignkey (RC2 - condition)))
...
Rn = group-select
       (RCn (quantifierN foreignkey (RCn - condition)))
R[n+1] = R1 intersect/union R2 intersect/union ...
                            intersect/union Rn
R[n+2] = select (RP (RP - condition))
R[n+3] = project (R[n+2] (RPprimekey))
R[n+4] = R[n+1] (RPprimekey) join RP (primekey)
R[n+5] = project (R[n+4] (RPprimekey))
R[n+6] = R[n+5] intersect/union R[n+3]
R[n+7] = R[n+6](RPprimekey) join RP (primekey)
R[n+8] = project (R[n+7] (*))
```

1b) Multiple Level Retrieval.

When a grandparent, parent and child are involved in a query we deal with two levels of nested quantifier expressions. As an example, consider the query:

Get full details of an Airline with headquarters located in San Diego where most of its aircraft have (a) Boeing as a manufacturer and (b) at least 1 scheduled 'computer repair' service and (c) all their necessary parts installed on board the plane.

The equivalent COOL expression is:

```
select * from Airline
where hqadd = ''San Diego''
and for most Airline's Aircraft ( fabricant = "Boeing"    LEVEL (1)
      and for at least 1 Aircraft's Service               LEVEL (2)
                     ( description = ''computer repair'')
      and for all Aircraft's Parts_on_Board               LEVEL (2)
                     (status = ''ON''));
```

The equivalent ERA routine is:

```
R0 = group_select
     (Service (for at least 1 AC# (description = "computer repair")))
R1 = group_select (Parts_on_Board (for all AC# (status = "ON")))
R2 = R0 intersect(AC#) R1
R3 = R2 (AC#) pjoin (p0) Aircraft (AC#)
R4 = group_select (R3 (for most AL# ((fabricant = "Boeing") AND p0)))
R5 = select ( Airline ( hqadd = "San Diego"))
R6 = project ( R5 ( AL#))
R7 = R4 (AL#) join Airline (AL#)
R8 = project ( R7 ( AL#))
R9 = R8 intersect(AL#) R6
```

```
R10 = R9 (AL#) join Airline (AL#)

R11 = project ( R10 (*))
```

The general case of a multiple level retrieval is presented later on in this Chapter.

2. Composite Parent-to-Child Genitive Relations (Grandparent Parent Child).

Here we assume that the parent does not have any condition on its attributes, which is invariably the case.

2a) One level expression.

Consider the example:

Name the location of each airline with more than 3000 employees where all the on board parts of all the aircraft of the airline have the status ON.

The equivalent COOL expression is:

```
select hqadd from Airline

where emp_num > 3000

and for all Airline's Aircraft's Parts_on_Board (status = ``ON'');
```

The composite parent-to-child genitive relation in the above example is *Airline's Aircraft's Parts_on_Board.*

The corresponding ERA routine is:

```
RR0 = project (Aircraft (AC#, AL#))

R1 = RR0 (AC#) join Parts_on_Board (AC#)

R2 = group_select (R1 (for all AL# (status = "ON")))

R3 = select ( Airline ( emp_num > 3000))
```

```
R4 = project ( R3 ( AL#))

R5 = R2 (AL#) join Airline (AL#)

R6 = project ( R5 ( AL#))

R7 = R6 intersect(AL#) R4

R8 = R7 (AL#) join Airline (AL#)

R9 = project ( R8 (hqadd))
```

An Aircraft can NOT have a condition in such a query. If it does, it is a query of the type already covered.

2b) Multiple levels expression.

Consider the example:

Name the location of each airline with more than 3000 employees for which, on all its aircraft, each of its (owned) maintenance-depot services (a) involves computer repairs, and (b) is offered in Calgary.

The equivalent COOL expression is:

```
select hqadd from Airline
where emp_num > 3000
and for all Airline's Aircraft's Service
     (description = ''computer repair''
      and for its Service's Maintenance_Depot
           (address = "Calgary"));
```

The genitive relations in the example above are: (1) the composite parent-to-child one *Airline's Aircraft's Service*, and (2) the simple child-to-parent one *Service's Maintenance_Depot*.

The corresponding ERA routine is:

```
R0 = select (Maintenance_Depot (address = "Calgary"))

R1 = project (R0 (MD#))

R2 = R1 (MD#) join Service (MD#)

R3 = project (R2 (SV#))

R4 = R3 (SV#) pjoin (p0) Service (SV#)

RR5 = project (Aircraft (AC#, AL#))

R6 = RR5 (AC#) join R4 (AC#)

R7 = group_select

      (R6 (for all AL# ((description = "computer repair") AND p0)))

R8 = select ( Airline ( emp_num > 3000))

R9 = project ( R8 ( AL#))

R10 = R7 (AL#) join Airline (AL#)

R11 = project ( R10 ( AL#))

R12 = R11 intersect(AL#) R9

R13 = R12 (AL#) join Airline (AL#)

R14 = project ( R13 (hqadd))
```

6.5.4 Child-to-Parent Genitive Relations

1. Simple child-parent genitive relation.

1a) One level query.

Consider we take the retrieval:

Get the fabricant of the aircraft owned by British Airways.

The COOL expression is:

```
select fabricant from Aircraft

where for its Aircraft's Airline ( AL# = ''BA'');
```

The simple child-to-parent genitive relation in the example above is *Aircraft's Airline*.

The above COOL expression can be reduced to the following extended relational algebra (ERA) routine:

```
R0 = select (Airline (AL# = "BA"))

R1 = project (R0 (AL#))

R2 = R1 (AL#) join Aircraft (AL#)

R3 = project (R2 (AC#))

R4 = R3 (AC#) join Aircraft (AC#)

R5 = project ( R4 (fabricant))
```

1b) Multiple level expression.

Consider the example:

Get the description of each service performed for British Airways aircraft for which the majority of parts on board are not mounted.

The equivalent COOL expression is:

```
select description from Service

where for its Service's Aircraft ( AL# = ''BA'' and

for most Aircraft's Parts_on_Board (status = ''OFF''));
```

The genitive relations in the above example are: (1) the simple child-to-parent one *Service's Aircraft*, and (2) the simple parent-to-child one *Aircraft's Parts_on_Board.*

The ERA routine is:

```
R0 = group_select (Parts_on_Board (for most AC# (status = "OFF")))

R1 = select (Aircraft (AL# = "BA"))

R2 = project (R1 (AC#))

R3 = R2 intersect(AC#) R0

R4 = R3 (AC#) join Service (AC#)

R5 = project (R4 (SV#))

R6 = R5 (SV#) join Service (SV#)

R7 = project ( R6 (description))
```

2. Composite child-to-parent genitive relation (or Child Parent Grand-parent).

2a) Single level expression.

Suppose we take the retrieval:

Get the description of each job in service projects that involve computer repair services.

The COOL expression is:

```
select descr from Job

where for its Job's Service_Project's Service

                    ( description = ''computer repair'');
```

The composite child-to-parent genitive relation in the above example is *Job's Service_Project's Service.*

The ERA routine for the above query will be:

```
R0 = select (Service (description = "computer repair"))
```

```
R1 = project (R0 (SV#))

R2 = R1 (SV#) join Service_Project (SV#)

R3 = project (R2 (SVP#))

R4 = R3 (SVP#) join Job (SVP#)

R5 = project (R4.(J#))

R6 = R5 (J#) join Job (J#)

R7 = project ( R6 (descr))
```

2b) Multiple level expression.

Suppose we take the retrieval:

Get the description of each job in service projects that involve computer repair services carried out on Alitalia aircraft.

The COOL expression is:

```
select descr from Job

where for its Job's Service_Project's Service

      ( description = ''computer repair''

        and for its Service's Aircraft (AL# =''AL''));
```

The genitive relations in the above query are: (1) the composite child-to-parent one *Job's Service_Project's Service*, and (2) the simple child-to-parent one *Service's Aircraft*.

The ERA routine for the above query will be:

```
R0 = select (Aircraft (AL# = "AL"))

R1 = project (R0 (AC#))

R2 = R1 (AC#) join Service (AC#)
```

```
R3 = project (R2 (SV#))

R4 = select (Service (description = "computer repair"))

R5 = project (R4 (SV#))

R6 = R5 intersect(SV#) R3

R7 = R6 (SV#) join Service_Project (SV#)

R8 = project (R7 (SVP#))

R9 = R8 (SVP#) join Job (SVP#)

R10 = project (R9 (J#))

R11 = R10 (J#) join Job (J#)

R12 = project ( R11 (descr))
```

6.5.5 The General Case - Hierarchical COOL expressions with n levels of nested quantifier expressions

The above queries can be generalized to n levels of nested quantifier expressions by induction. Suppose we extend the hierarchy RP, relation parent and the child relations RC1 and RC2, presented in Section 6.5.3, as follows. The parent relation RP has n children, the relations (RC1, RC2, ..., RCn). Relation RC1 has the children (RC11, RC12,..., RC1n), relation RC2 has the children (RC21, RC22,..., RC2n), so on, and finally relation RCn has the children (RCn1, RCn2, ..., RCnn), and then each of these children can have their own n children e.g. RC11 has children RC111, RC112,..., RC11n, and so on.

Figure 6.5 shows a graphical representation of this hierarchy.

The general retrieval is:

Retrieve each RP tuple for which RP - condition (compound condition involving

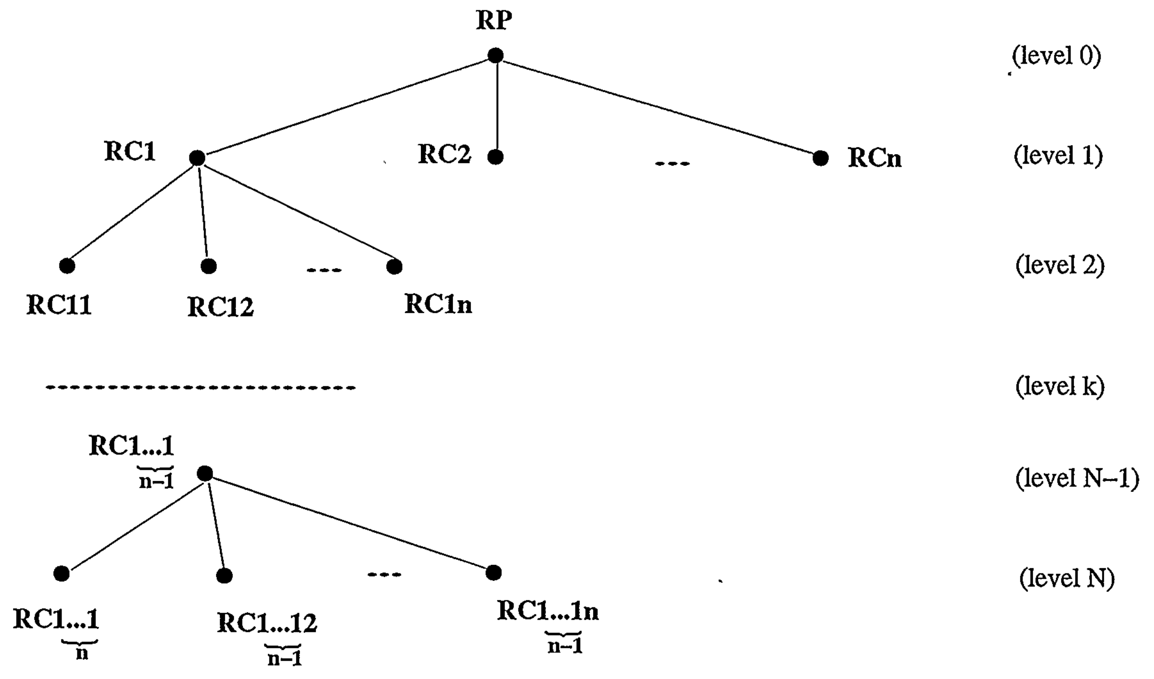

Figure 6.5: A hierarchy of relations with N+1 levels and each node has n offsprings

RP's fields) holds and for which a specific quantity of related RC1 tuples obey RC1 - condition (compound condition involving RC1's fields) and so on for the RP's n children. For the specific quantity of related RC1 tuples a specific quantity of related RC11 tuples obey RC11 - condition and so on for RC1's n children. For the specific quantity of related RC11 tuples a specific quantity of related RC111 tuples obey RC111 - condition and so on.

The hierarchy grows in two dimensions (vertical and horizontal).

The general COOL query looks like:

```
select * from RP
where (RP - cond)
and/or q1 RP's RC1
```

```
                          (RC1-cond

                           and/or q11 RC1's RC11

                                          (RC11-cond

                                           and/or q111 RC11's RC111

                                                             (RC111-cond

                                                              and/or ...)

                                           and/or q112 RC11's RC112

                                                             (...)

                                           ...)

                           and/or q12 RC1's RC12 (...)

                           ...

                           and/or q1n RC1's RC1n (...))

     and/or q2 RP's RC2

                          (RC2-cond

                           and/or ...)

     ...

     and/or qn RP's RCn

                          (RCn-cond

                           and/or qn1 RCn's RCn1

                                          (RCn1-cond

                                           and/or qn11 RCn1's RCn11

                                                             (RCn11-cond

                                                              and/or ...)

                                           and/or qn12 RCn1's RCn12
```

```
                                                    (...)
                                          ...)
                           and/or qn2 RC's RCn2
                                         (...))
```

q1 stands for quantifier1, q2 stands for quantifier2, and so on. The above query can be broken up into the hierarchy for RC1, the hierarchy for RC2, and so on. If R1, R2,... are the result of evaluating each hierarchy, the corresponding general ERA routine, for a query involving the data structure from figure 6.5, can fit into the ERA routine referred to as ERA-Gen1 in Section 6.5.3.

```
R1 = ...
R2 = ...
...
Rn = ...
R[n+1] = R0 intersect/union R1 intersect/union ...
                                   ... intersect/union Rn
R[n+2] = select (RP (RP - condition))
R[n+3] = project (R[n+2] (RPprimekey))
R[n+4] = R[n+1] (RPprimekey) join RP (primekey)
R[n+5] = project (R[n+4] (RPprimekey))
R[n+6] = R[n+5] intersect/union R[n+3]
R[n+7] = R[n+6](RPprimekey) join RP (primekey)
R[n+8] = project (R[n+7] (*))
```

The R1, R2, ..., Rn ERA expressions have similar corresponding queries, such as:

The general COOL expression for the child RC1 hierarchy:

```
select primary_key from RP
where q1 RP's RC1
          (RC1-cond
           and/or q11 RC1's RC11
                       (RC11-cond
                        and/or q111 RC11's RC111
                                     (RC111-cond
                                      and/or ...
                                      ...
                                      and/or q$\underbrace{1...1}_{n}$ RC$\underbrace{1...1}_{n-1}$'s RC$\underbrace{1...1}_{n}$
                                             (RC$\underbrace{1...1}_{n}$-cond))
                        and/or q112 RC11's RC112(...)
                        ...
                        and/or q11n RC11's RC11n(...))
           and/or q12 RC1's RC12 (...)
           ...
           and/or q1n RC1's RC1n (...))
```

The general ERA routine for the child RC1 hierarchy is:

```
level N:  R[N,1] = group-select
            (RC$\underbrace{1...1}_{n}$ (q$\underbrace{1...1}_{n}$ foreignkey (RC$\underbrace{1...1}_{n}$-cond)))
          ...
```

```
                R[N,n] = group-select
                   (RC$\underbrace{1\ldots1}_{n-1}$n (q$\underbrace{1\ldots1}_{n-1}$n foreignkey (RC$\underbrace{1\ldots1}_{n-1}$n-cond)))
                R[N,C1N-1]= R[N,1] intersect/union R[N,2] ...
                          ... intersect/union R[N,n]
                R[N] =
                   R[N,C1N-1](foreignkey) pjoin(p) RC$\underbrace{1\ldots1}_{n-1}$(primekey)
      level N-1:R[N-1,1] = group-select
                   (R[N] (q$\underbrace{1\ldots1}_{n-1}$ foreignkey (RC$\underbrace{1\ldots1}_{n-1}$-cond and/or p)))
```

We repeat the above sequence for the remaining children in level N-1: RC$\underbrace{1\ldots1}_{n-2}$2, …, RC$\underbrace{1\ldots1}_{n-2}$n, giving the results: R[N-1,2], …, R[N-1,n]. Further on, level N-1 will give R[N-1] as final result, and the same algorithm will be repeated until level 0 will be reached.

```
                ...
                R[N-1,C1N-2]= R[N-1,1] intersect/union R[N-1,2] ...
                          ... intersect/union R[N-1,n]
                R[N-1] =
                   R[N-1,C1N-2](foreignkey) pjoin(p) RC$\underbrace{1\ldots1}_{n-2}$(primekey)
      level N-2:R[N-2,1] = group-select
                   (R[N-1] (q$\underbrace{1\ldots1}_{n-2}$ foreignkey (RC$\underbrace{1\ldots1}_{n-2}$-cond and/or p)))
                ...
                R[k+1,C1k]= R[k+1,1] intersect/union R[k+1,2] ...
                          ... intersect/union R[k+1,n]
                R[k+1] =
                 R[k+1,C1k] (foreignkey) pjoin(p) RC$\underbrace{1...1}_{k}$(primekey)
      level k:  R[k,1] = group-select
                  (R[k+1] (q$\underbrace{1...1}_{k}$ foreignkey (RC$\underbrace{1...1}_{k}$-cond and/or p)))
                ...
                R[2,C11]= R[2,1] intersect/union R[2,2] ...
                         ... intersect/union R[2,n]
                R2 = R[2,C11](foreignkey) pjoin(p) RC1 (primekey)
      level 1:  R[1,1] = group-select
                  (R2 (q1 foreignkey (RC1-cond and/or p)))
```

Repeat for all the children from level 1 and get R[1,2] ... R[1,n]

```
                ...
                R[1,C0]= R[1,1] intersect/union R[1,2] ...
                         ... intersect/union R[1,n]
                R1 = select (RP (RP-cond))
      level 0:  R0 = R[1,C0] (foreignkey) join RP(primekey)
                R[01] = R0 intersect/union R1
                RR = project (R01 (*))
```

R[N,i] is the result of evaluating a quantified genitive relation expression where i=1 to n specifies the children of a relation node. N is the level number of the children.

R[k+1,C1k] is the result of evaluating the compound logical expression consisting of quantified genitive relation expressions involving the children of node 1 from level k (node RC$\underbrace{1\ldots1}_{n-1}$), indicated by C1k. In this case the level of the children is k+1, and is the first subscript of R.

R[N] is the result of an outer join (or pjoin) between $RC\underbrace{1\ldots1}_{n-1}$, the first node on level N-1, as a parent, and the relation that captures the relationships with its children from level N, R[N,C1N-1].

6.5.6 The General Case - Network COOL expressions with n levels of nested quantifier expressions

The routine above is the ERA routine corresponding to a general COOL query involving a hierarchy. It requires only a few changes to make the above routine work for a COOL retrieval involving a network. This means that instead of just parent-to-child genitive relations (or 1:n relationships) we can also have child-to-parent genitive relations (or n:1 relationships). The following are the possible changes that can be applied to the hierarchy routine above, and for each of them we derive the correspondent ERA in it.

(1) Child with condition. If a child involved in a parent-to-child genitive relation has a condition (see), the operation 'group-select' from R[level,i], applied to the child becomes 'subgroup-select'.

(2) Composite parent-to-child genitive relation. If instead of a parent-to-child genitive relation we have a composite parent-to-child genitive relation, of the form: Grandparent Parent Child, the corresponding R[level,i] will be changed to the following sequence:

```
...
R[k+1,C1k]= R[k+1,1] intersect/union R[k+1,2] ...
          ... intersect/union R[k+1,n]
```

```
            R[k+1] =
              R[k+1,C1k](Child_primekey) pjoin(p) Child (primekey)
level k:    R[k0] = project(Parent(primekey, foreignkey))
            R[k1] =
             R[k+1] (Child_foreignkey) join R[k0] (Parent_primekey)
            R[k,i] = group_select
              (R[k1] (quantifier Grandparent_primekey
                                        (Child_cond and/or p)))
```

Relations R[k0] and R[k1] are giving intermediate results.

(3) Child-to-parent genitive relation. If instead of a parent-to-child genitive relation we have a simple child-to-parent genitive relation, of the form: Child's Parent, the corresponding R[level,i] will be changed to the following sequence:

```
            ...
            R[k+1,C1k]= R[k+1,1] intersect/union R[k+1,2] ...
                        ... intersect/union R[k+1,n]
level k:    R[k0] = select (Parent (Parent_cond))
            R[k1] = project (R[k0] (Parent_primekey))
            R[k2] = R[k1] intersect/union R[k+1,C1k]
            R[k3] = R[k2] (Parent_primekey) join Child (foreignkey)
            R[k,i] = project (R[k3] (Child_primekey))
```

Relations R[k0], ..., R[k3] are giving intermediate results.

(4) Composite child-to-parent genitive relation. If instead of a parent-to-child genitive relation we have a composite child-to-parent genitive relation, of the form: Child Parent Grandparent, the corresponding R[level,i] will be changed to the following sequence:

```
                ...
                R[k+1,C1k]= R[k+1,1] intersect/union R[k+1,2] ...
                          ... intersect/union R[k+1,n]
     level k:   R[k0] = select (Grandparent (Grandparent_cond))
                R[k1] = project (R[k0] (Grandparent_primekey))
                R[k2] = R[k1] intersect/union R[k+1,C1k]
                R[k3] =
                 R[k2] (Grandparent_primekey) join Parent (foreignkey)
                R[k4] = project (R[k3] (Parent_primekey))
                R[k5] = R[k4] (Parent_primekey) join Child (foreignkey)
                R[k,i] = project (R[k5] (Child_primekey))
```

Relations R[k0], ..., R[k5] are giving intermediate results.

6.5.7 Recursive Queries

Suppose we take the recursive query:

Get the quantity of parts directly containing at least 4 cogs that each have at least 2 P2 parts inside.

The COOL expression is:

```
select qty from PartType_Inventory
where for at least 4 PartType_Inventory's *inner_parts Construct
(for its Construct's PT_inner PartType_Inventory
   (typename="cog"
    and for at least 2 PartType_Inventory's *inner_parts Construct
                                                       (PT_inner="P2")));
```

The ERA routine for the above query will be:

```
R0 = group_select
       (Construct (for at least 2 PT_outer (PT_inner = "P2")))
R1 = select (PartType_Inventory (typename = "cog"))
R2 = project (R1 (PT#))
R3 = R2 intersect(PT#) R0
R4 = R3 (PT#) join Construct (PT_inner)
R5 = project (R4 (C#))
R6 = R5 (C#) pjoin (p0) Construct (C#)
R7 = group_select (R6 (for at least 4 PT_outer (p0)))
R8 = R7 (PT#) join PartType_Inventory (PT#)
R9 = project ( R8 (qty))
```

This query expression is quite intricate and appears complex even in COOL. Nevertheless the corresponding SQL expression is much more complex.

6.5.8 Translation of COOL DML expressions to SQL

INSERT

The system performs only instance by instance insert.

The syntax is close to SQL, and looks like:

```
<insert_instance_statement> ::=
                 INSERT OBJECT INSTANCE INTO <objectcls>
                        '(' <insert_value_commalist> ')'
<insert_value_commalist> ::=
                 <insert_value>
              |  <insert_value_commalist> ',' <insert_value>
<insert_value> ::=
               <attribute> ':' <alpha>
             | <attribute> ':' <numeric>
```

Example:

```
insert obj ins into Airline
       ( AL# : "RO", hqadd : "Bucharest", emp_num : 3000);
insert obj ins into Aircraft
       ( AC# : "AB41", AL# : "AF", fabricant : "Boeing", type :"B747");
```

For each 'insert instance' statement, the following operations will be executed:

```
* generate unique OID within the mother class and update 'cool_objects'
for each attribute
        if attribute is a superkey
        then ///  create/update the correspondent reflist  ///
             for each read correspondent reference list from cool_keys
                * get the name of the parent of the reflist
                * get the OID of the parent instance
                if (parent_child_reflist exists)
                then
                  *add a new record with the OID of the parent and
                   OID of the child
                else
                  *create the relationship table
                  *add a new record with the OID of the parent and
                   OID of the child
                endif
             end for


        endif
        * write attribute in an SQL format
```

An 'insert' statement for many instances that can populate an empty object class is a convenient additional feature, that requires no research to add. It is not vital for the prototype. Also, to populate a class from a intermediate result table requires access to each instance, one instance at a time, which means file processing and not the table processing of a relational database. Two instances with the same primary key can be inserted in the database, but they will be asigned different OIDs. An error message will be generated if a child instance is going to be inserted but the parent instance is not found in the database.

UPDATE

The update statement implemented is a multiple-instance update and the syntax is very close to SQL.

Here is the syntax:

```
<update_statement> ::=
   UPDATE <objectcls> WHERE <condition> SET <update_value_commalist>
<update_value_commalist> ::=
        <update_value>
     |  <update_value_commalist> ',' <update_value>
<update_value> ::=
        <attribute> ':' <scalar_expr>
     |  <attribute> ':' <alpha>
```

where <condition> is a boolean expression, <scalar_expr> is a relational expression with numeric values and <alpha> is a STRING.

Example:

```
update Airline where AL# = "RO" set hqadd : "arad";
```

DELETE

Delete statement has 2 versions:

1. Delete all instances (unconditioned) of a class. It is performed only if there are no children instances involved, and
2. Delete selected instances (or with condition). It is performed also only if there are no descendants.

An error message will be generated if instances with descendants are attempted to be deleted.

Syntax for unconditioned delete:

```
<delete_uncond_statement> :=
                                    DELETE ALL FROM <objectcls>
```

Syntax for conditioned delete:

```
<delete_uncond_statement> :=
                                    DELETE FROM <objectcls> WHERE <condition>
```

where <condition> is a boolean expression.

The implementation of a delete for composite objects (this includes deleting one or more instances and deleting an entire hierarchy if the instances have children) was not implemented because it does not raise any essential problems, and is only a programming exercise. The algorithm with embedded Sybase commands for this is:

```
procedure: gen_del_compobj
input: class_name, condition
start
  //  check if there are any descendants //
   * call sybase process to execute the query:
            ''select * from cool_keys
             where class_name = '%s'
             and key_type = 'RL' ''
  if (query == SUCCEED)
     * bind 'key_name' to the variable 'reflist_name'
     while ( there is a reflist )
```

```
       * get the name of the child class with key_type ='SK reflist'
       * get the name of the relationship table
                                                   'parent_child_reflist'
       * execute "select * from class_name" and get one by
                                                     one the OIDs.
       for each parent OID
            * get the children OIDs from the relationship table
            * list children one by one
  for each child OID
             * ask the user if he wants it deleted
          endfor
       endfor
     endwhile


  else
       * print error message "error in  cool_keys"


  endif
  if NO descendants for class_name
      // delete the instances of the parent and its links
         to its parent, from the table 'parent_child_reflist' //
       * execute ``delete from %s where %s", class_name, condition''
end.
```

6.5.9 Algorithms for translation of COOL queries to ERA

The algorithm for translation of a COOL query to ERA follows the parsing of the query, that is, the derivation tree. The leaves of the derivation or parse tree, read from left to right, form a string of characters that is nothing else but the input query. When, in the process of parsing, the end of a grammar rule is reached, the rule is reduced (right side of the rule is replaced by the left side) and, at the same time, some actions are performed by the parser. The actions can be generation of ERA routines or only saving of useful information for future reductions of grammar rules.

Consider a general select statement with n levels of nested quantified expressions. The BNF form of the select statement is as follows.

```
<select_statement> ::=
   SELECT <selection> FROM <object_class> [WHERE <where_expression>]
<selection> ::=
            <attribute_commalist>
         |  *
<where_expression> ::=
            <condition> [<logical_xref_list>]
         |  <quantified_xreference> [<logical_xref_list>]
<logical_xref_list> ::=
            <logical_xref>
         |  <logical_xref_list> <logical_xref>
<logical_xref> ::=
            OR <quantified_xreference>
```

```
          |  AND <quantified_xreference>
<quantified_xreference> ::=
            <quantified_genitive_relation> ( <where_expression> )
<quantified_genitive_relation> ::=
// Parent Child genitive relation where
   (1) is the formal syntax, and  //
        <quantifier_pc> <parent> . <reference_attribute>
                                               * <formal_child>
// (2) is the natural language syntax //
      |  <quantifier_pc> <parent>'S
                        [* <reference_attribute>] <natural_child>
// Composite Parent Child genitive relation where
   (1) is the formal syntax, and //
      |  <quantifier_pc> <grandparent> . <reference_attribute>
                     * <parent> . <reference_attribute> * <child>
// (2) is the natural language syntax //
      |  <quantifier_pc> <grandparent>'S
                     [* <reference_attribute>] <parent>'S
                               [* <reference_attribute>] <child>
// Child Parent genitive relation where
   (1) is the formal syntax, and //
      |  <quantifier_cp> <child> . <child_superkey> * <parent>
// (2) is the natural language syntax //
      |  <quantifier_cp> <child>'S [<child_superkey>] <parent>
```

```
// Composite Child Parent genitive relation where
   (1) is the formal syntax, and //
     |  <quantifier_cp> <child> . <child_superkey> * <parent>
                                  . <parent_superkey * <grandparent>
// (2) is the natural language syntax //
     |  <quantifier_cp> <child>'S [<child_superkey>] <parent>'S
                                    [<parent_superkey>] <grandparent>
<child> ::=
        <object_class>
<parent> ::=
         <object_class>
<grandparent> ::=
         <object_class>
<formal_child> ::=
              <object_class>
            |  ( <object_class> ( <condition> ) )
<natural_child> ::=
              <object_class>
            |  ( <condition> ) <object_class>
<quantifier_pc> ::=
               FOR ALL
             | FOR MOST
             | FOR NONE
             | FOR ALL BUT INTNUM
             | FOR AT MOST INTNUM
             | FOR AT LEAST INTNUM
             | FOR MORE THAN INTNUM
             | FOR LESS THAN INTNUM
             | FOR EXACTLY INTNUM
             | FOR NOT INTNUM
<quantifier_cp> ::=
               FOR THE
             | FOR ITS
             | FOR HER
             | FOR HIS
```

If we combine the rules for the nonterminals <where_expression>, <logical_xref_list>, and <logical_xref> from above, we obtain:

```
<where_expression> ::=
                  [[<condition>] AND/OR] <quantified_xreference>
                                 AND/OR <quantified_xreference>
                                 ...
```

where <condition> is a boolean expression.

If in the rule for the <quantified_xreference> we replace <quantified_genitive_relation> with <quantifier> <genitive_relation>, we obtain:

```
<quantified_xreference> ::=
             <quantifier> <genitive_relation> [(<where_expression>)]
```

In this way we emphasize the <genitive_relation> nonterminal, that can have four different types, such as simple parent-to-child or <pc_gen_rel>, composite parent-to-child or <pc_composite_gen_rel>, simple child-to-parent or <cp_gen_rel>, and composite child-to-parent or <cp_composite_gen_rel>, as it is shown in the rule bellow.

```
<genitive_relation> :=
                        <pc_gen_rel>
                      | <pc_composite_gen_rel>
                      | <cp_gen_rel>
                      | <cp_composite_gen_rel>
```

Each of the four types of the genitive relation has two syntactic expressions: a formal one and a natural one. The natural one is close to the genitive case expression from the English language.

For each of the genitive relations listed above we can have three cases of <where_expression>.

```
(a)  <where_expression> ::=
                             <condition>
(b)  <where_expression> ::=
                             <condition> AND/OR <relation_result_qexpr>
    // <relation_result_qexpr> is the relation result of a
           quantified_expression and has only one column  //
(c)  <where_expression> ::=
                             <relation_result_qexpr>
```

The algorithm for the code generation phase of the translation of the COOL nested quantifier expressions to ERA routines, is given in the procedure 'quantified_xref'. This procedure has two input arguments : the quantified_genitive_relation and the where_expression. The two arguments could take the values described above.

```
procedure quantified_xref (genitive_relation, where_expression)
{
// Parent to Child genitive_relation  //
if (genitive_relation = 'Parent'S [reference_list] Child') then


        // Case (a) //
    if (where_expression = condition) then
       if (reference_attribute is not null) then
           if (Child has condition) then
              *generate: R(i) = subgroup_select
              (Child (quantifier_pc
                (child_superkey child_condition) (condition)))
           else
              *generate: R(i) = group_select
              (Child (quantifier_pc child_superkey
                                             (condition)))
           endif
       else
           if (Child has condition) then
    *generate: R(i) = subgroup_select
            (Child (quantifier_pc (parent_primekey child_condition)
                                                       (condition)))
        else
            *generate: R(i) = group_select
            (Child (quantifier_pc parent_primekey (condition)))
        endif
    endif
    link = parent
    return (``R(i) link'')
endif


    // Case (b) //
if (where_expression = condition AND/OR quantified_xreference)
    //  R(p) is the relation result of the previous level of
    quantified_expression, that is R(p)=quantif_xreference(...) //
then
    *execute quantified_xreference(): return R(p) link
    *check if Child = link  // semantic check //
    *generate: R(i) =
          R(p) (child_primekey) pjoin(p(j)) Child (child_primekey)
       if (Child has condition) then
          if (reference_attribute exists) then
             *generate: R(i+1) = subgroup_select
```

```
              (R(i) (quantifier_pc (child_superkey child_cond)
                                    (condition AND/OR p(j))))
          else
             *generate: R(i+1) = subgroup_select
              (R(i) (quantifier_pc (parent_primekey child_cond)
                                    (condition AND/OR p(j))))
          endif
       else
          if (reference_attribute exists) then
             *generate: R(i+1) = group_select
             (R(i) (quantifier_pc child_superkey
                                    (condition AND/OR p(j))))
          else
             *generate: R(i+1) = group_select
             (R(i) (quantifier_pc parent_primekey
                                    (condition AND/OR p(j))))
          endif
       endif
    link = parent
    return (``R(i+1) link'')
 endif

    // Case (c) //
 if (where_expression = quantified_xreference)
```

```
       //  R(p) is the relation result of the previous level of
       quantified_expression, that is R(p)=quantif_xreference(...) //
    then *follow the steps from case (b) with one exception: in the
          (sub)group_select expressions, the field
           'condition AND/OR p(j)' will be replaced by 'p(j)'
endif


// composite parent to child genitive_relation   //
if (genitive_relation =
              'Grandparent'S [grandpar_reference_list] Parent'S
                                     [par_reference_list] Child')
then


       // Case (a) //
    if (where_expression = condition) then
        if (grandpar_reference_list is not null) then
            par_foreignkey = get_foreignkey
                                     (parent, grandpar_reference_list)
        else
            par_foreignkey = grandpar_primekey
        endif
        *generate: RR(i) =
              project (Parent (par_primekey, par_foreignkey))
        if (par_reference_list is not null) then
```

```
            child_foreignkey = get_foreignkey
                                           (child, par_reference_list)
        else
            child_foreignkey = par_primekey
        endif
        *generate: R(i+1) =
                RR(i) (par_primekey) join Child (child_foreignkey)
        *generate: R(i+2) =
                group_select (R(i+1)
                       (quantifier_pc par_foreignkey (condition)))
        link = grandparent
        return (``R(i+2) link'')
    endif


       // Case (b) //
    if (where_expression = condition AND/OR quantified_xreference)
       //  R(p) is the relation result of the previous level of
       quantified_expression, that is R(p)=quantif_xreference(...) //
    then
        *execute quantified_xreference(): return (``R(p) link'')
        *check if Child = link // semantic check //
        *generate: R(i) =
            R(p) (child_primekey) pjoin (p(j)) Child (child_primekey)
        if (grandpar_reference_list is not null) then
```

```
            par_foreignkey = get_foreignkey
                                   (parent, grandpar_reference_list)
        else
            par_foreignkey = grandpar_primekey
        endif
        *generate: RR(i+1) = project
                        (Parent (par_primekey, par_foreignkey))
        if (par_reference_list is not null) then
            child_foreignkey = get_foreignkey
                                   (child, par_reference_list)
        else
            child_foreignkey = par_primekey
        endif
        *generate: R(i+2) =
              RR(i+1) (par_primekey) join R(i) (child_foreignkey)
        *generate: R(i+3) = group_select
              (R(i+2) (quantifier_pc par_foreignkey
                                         (condition AND/OR p(j))))
        link = grandparent
        return (''R(i+3) link'')
    endif

       // Case (c) //
    if (where_expression = quantified_xreference)
```

```
       //  R(p) is the relation result of the previous level of
       quantified_expression, that is R(p)=quantif_xreference(...) //
    then *follow the steps from case (b) with one exception: in the
          (sub)group_select expressions, the field
          'condition AND/OR p(j)' will be replaced by 'p(j)'
endif


// Child to Parent genitive_relation  //
if (genitive_relation = 'Child'S [child_superkey] Parent') then


       // Case (a) //
    if (where_expression = condition) then
       *generate: R(i) = select (Parent (condition))
       *generate: R(i+1) = project (R(i) (par_primekey))
       if (child_superkey is null) then
          child_superkey =  par_primekey
       *generate: R(i+2) =
            R(i+1) (par_primekey) join Child (child_superkey)
       *generate: R(i+3) = project (R(i+2) (child_primekey))
       link = child
       return (''R(i+3) link'')
    endif


       // Case (b) //
```

```
if (where_expression = condition AND/OR quantified_xreference)
   //  R(p) is the relation result of the previous level of
   quantified_expression, that is R(p)=quantif_xreference(...) //
then
   *execute quantified_xreference(): return R(p) link
   *check if Parent = link  // semantic check //
   *generate: R(i) = select (Parent (condition))
   *generate: R(i+1) = project (R(i) (par_primekey))
   *generate: R(i+2) = R(i+1) intersect/union R(p)
   if (child_superkey is null) then
      child_superkey =  par_primekey
   *generate: R(i+3) =
        R(i+2) (par_primekey) join Child (child_superkey)
   *generate: R(i+4) = project (R(i+3) (child_primekey))
   link = child
   return (``R(i+4) link'')
endif


   // Case (c) //
if (where_expression = quantified_xreference)  then
   //  R(p) is the relation result of the previous level of
   quantified_expression, that is R(p)=quantif_xreference(...) //
   *execute quantified_xreference(): return (``R(p) link'')
   *check if Parent = link  // semantic check //
```

```
       if (child_superkey is null) then
          child_superkey =  par_primekey
       *generate: R(i) =
             R(p) (par_primekey) join Child (child_superkey)
       *generate: R(i+1) = project (R(i) (child_primekey))
       link = child
       return (``R(i+1) link'')
    endif
endif


// composite Child to Parent genitive_relation   //
if (genitive_relation = 'Child'S [child_superkey] Parent'S
[parent_superkey] Grandparent')
then


       // Case (a) //
    if (where_expression = condition) then
       *generate: R(i) = select (Grandparent (condition))
       *generate: R(i+1) = project (R(i) (grandpar_primekey))
       if (parent_superkey is null) then
          parent_superkey =  grandpar_primekey
       *generate: R(i+2) =
          R(i+1) (grandpar_primekey) join Parent (parent_superkey)
       *generate: R(i+3) = project (R(i+2) (par_primekey))
```

```
   if (child_superkey is null) then
      child_superkey =  par_primekey
   *generate: R(i+4) =
         R(i+3) (par_primekey) join Child (child_superkey)
   *generate: R(i+5) = project (R(i+4) (child_primekey))
   link = child
   return (''R(i+5) link'')
endif


   // Case (b) //
if (where_expression = condition AND/OR quantified_xreference)
then
   //  R(p) is the relation result of the previous level of
   quantified_expression, that is R(p)=quantif_xreference(...) //
   *execute quantified_xreference(): return (''R(p) link'')
   *check if Grandparent = link   // semantic check//
   *generate: R(i) = select (Grandparent (condition))
   *generate: R(i+1) = project (R(i) (grandpar_primekey))
   *generate: R(i+2) = R(i+1) intersect/union R(p)
   if (parent_superkey is null) then
      parent_superkey =  grandpar_primekey
   *generate: R(i+3) =
       R(i+2) (grandpar_primekey) join Parent (parent_superkey)
   *generate: R(i+4) = project (R(i+3) (par_primekey))
   if (child_superkey is null) then
      child_superkey =  par_primekey
   *generate: R(i+5) =
           R(i+4) (par_primekey) join Child (child_superkey)
   *generate: R(i+6) = project (R(i+5) (child_primekey))
   link = child
   return (``R(i+6) link'')
endif


   // Case (c) //
if (where_expression = quantified_xreference) then
   //  R(p) is the relation result of the previous level of
   quantified_expression, that is R(p)=quantif_xreference(...) //
   *execute quantified_xreference(): return R(p) link
   *check if Grandparent = link  // semantic check //
   *generate: R(i) =
    R(p) (grandpar_primekey) join Grandparent (grandpar_primekey)
   *generate: R(i+1) = project (R(i) (grandpar_primekey))
   if (parent_superkey is null) then
      parent_superkey =  grandpar_primekey
   *generate: R(i+2) =
     R(i+1) (grandpar_primekey) join Parent (parent_superkey)
   *generate: R(i+3) = project (R(i+2) (par_primekey))
   if (child_superkey is null) then
```

```
  child_superkey =  par_primekey
       *generate: R(i+4) =
           R(i+3) (par_primekey) join Child (child_superkey)
       *generate: R(i+5) = project (R(i+4) (child_primekey))
       link = child
       return (``R(i+5) link'')
    endif
endif
}
```

The algorithm above gives the code generation, that is, the ERA routine, when the derivation rule of the nonterminal <quantifier_expression> is reduced.

The following rule, which will be reduced in the translation process, is the derivation rule of the nonterminal <where_expression>.

```
<where_expression> ::=
                   [[<condition>] AND/OR] <quantifier_expression>
                                 AND/OR <quantifier_expression>...
```

We deal with three distinct cases:

```
(a) where_expression = condition
(b) where_expression = condition AND/OR quantified_xreference
(c) where_expression = quantified_xreference
```

For each of the cases in which a <where_expression> can appear, the following code will be generated:

```
Case(a)

        *generate: R(i) = select ( object_class ( condition))


Case(b)

        *check if object_class = link  // semantic check //

        *generate: R(i) = select ( object_class ( condition))

        *generate: R(i+1) = project ( R(i) ( primekey))

        *generate: R(i+2) =

                     object_class (primekey) join R(p) (primekey)

        *generate: R(i+3) = project ( R(i+2) ( primekey))

        *generate: R(i+4) = R(i+3) intersect/union R(i+1)

        *generate: R(i+5) =

                     R(i+4) (primekey) join object_class (primekey)


Case(c)

        *check if object_class = link  // semantic check //

        *generate: R(i) = object_class (primekey) join R(p) (primekey)
```

The last step in the code generation for a <select_statement> is the reduction of the following rule:

```
<select_statement> :=

   SELECT <selection> FROM <object_class> [WHERE <where_expression>]
```

Code generation in this case will be:

```
if (where_expression is null)
        *generate: R(i) = project ( object_class (A1, A2,...An))
else
        *generate: R(i) = project ( R(i-1) (A1, A2,...An))
```

where A1, A2,..., An are the attributes of the object_class.

6.5.10 Second step: Translation of ERA operations to SQL

This step was implemented as a separate compiler that reads an ERA file and translates it into SQL queries. The lex file for this step will contain the regular expressions for ERA's tokens and the associated actions, and the yacc file will contain the ERA's grammar production rules and the corresponding actions for code generation in SQL. Each ERA statement is converted to a set of SQL expressions. We look at them one by one.

1. **Join statement.** The statement refers to the natural join of conventional relational algebra. In the majority of the situations, 'join' operation is performed between a single-attribute relation, that contains the join attribute, and a regular relation. Thus, the general case, translates as follows.

   ERA:

   ```
   R = R1 (attribute_1) join R2 (attribute_2)
   ```

   Translates to:

   SQL:

```
select R2.* into #R
from R1, R2
where R1.attribute_1 = R2.attribute_2
```

In Sybase SQL [Syb91b] the '#' sign before a relation name means that the relation is temporary.

A special situation occurs with queries that contain composite parent-to-child genitive relations. In these cases we need to perform a natural join , using SQL, between a two-attribute relation and a regular relation, with join attributes having the same name. Sybase Transact-SQL, unfortunately, will not allow the result, with two columns of the same name, to be stored as a temporary relation; and we need such a temporary relation for further SQL processing. Sybase does this, because it has no natural join with SQL, only an equi-join. To solve this problem one solution would be to change the name of one of the common attributes before performing the equi-join on it. Another solution would be to specify the fields of each relation involved in the join in the SQL 'select' expression, with the exception of the one of the join-fields; avoiding in this way, the presence of two fields with the same name in the result of the equi-join. This second solution could introduce unnecessary complexity in the implementation because at the moment of the translation the temporary tables are not created in the database, and their attributes cannot be found in the Sybase system tables. Thus, the first solution was chosen. This solution also works with different design techniques, such as the same name for foreign and primary key, and different names for them. In order to change the name of the column of a Sybase table we need to created a new table with the same attribute definitions as the source table and then to copy the source table into the new table.

The operation of changing the name of a column is performed at the time when the two-column table is created by a 'project' operation (see section 'Project statement', later on).

We have named the newly created table as its join attribute, RRi, where i as an order number. The translation for the special situation described above is:

ERA:

```
R = RR1 (RR1) join R2 (attribute_2)
```

Translates to:

SQL:

```
select RR1.*, R2.* into #R
from RR1, R2
where RR1.RR1 = R2.attribute_2
```

2. **Select statement**

ERA:

```
R = select (R1 (condition))
```

Condition is a boolean expression involving attributes of relation R.

Translates to:

SQL:

```
select * into #R
from R1
where condition
```

3. **Group-Select statement.** The implementation of the group-select operation does not exactly correspond to the definition given in Chapter 5. To obtain an initial optimization of the ERA routine, the relation result of group-select does not contain all the fields in the child relation, but instead contains only the foreign key field. This simplification helps a lot in creating a connection bridge (the foreign key) between the different levels of the quantifier expressions (nested queries).

   ERA:

```
R = group-select
      (Child (quantifier_pc foreignkey (condition_expr)))
<condition_expr> ::=
                 <condition>
               | <condition> AND/OR <variable>
               | <variable>
<variable> ::=
           p(j)
```

From the Child and the foreignkey we can obtain the corresponding reflist (interrogating the table cool_keys). Knowing the reflist we can get the Parent. Reference_list names are unique in the database.

The equivalent SQL for the group-select operation corresponding to the natural quantifiers implemented are the following:

Quantifier_pc = "FOR ALL"

SQL:

```
select parent_primekey into #R
from Parent
where not exists
         (select *
          from Child
          where Parent.parent_primekey = Child.foreignkey
          and not (condition_expr) )
```

Quantifier_pc = "FOR MOST"

SQL:

```
select parent_primekey into #R
from Parent
where (select count (*)
       from Child
       where Parent.parent_primekey = Child.foreignkey
```

```
           and  (condition_expr) )
          >
          (select count (*)
           from Child
           where Parent.parent_primekey = Child.foreignkey
           and  not (condition_expr) )
```

Quantifier_pc = "FOR ALL BUT int"

SQL:

```
     select parent_primekey into #R
     from Parent
     where (select count (*)
            from Child
            where Parent.parent_primekey = Child.foreignkey
            and not (condition_expr) ) = int
```

General SQL expression for a family of quantifiers

```
Quantifier_pc = ''FOR NONE'', operation = ''='', int = 0
Quantifier_pc = ''FOR AT MOST int'', operation = ''<=''
Quantifier_pc = ''FOR AT LEAST int'', operation = ''>=''
Quantifier_pc = ''FOR MORE THAN int'', operation = ''>''
Quantifier_pc = ''FOR LESS THAN int'', operation = ''<''
Quantifier_pc = ''FOR EXACTLY int'', operation = ''=''
Quantifier_pc = ''FOR NOT int'', operation = ''<>''
```

All the above quantifiers can be translated with the following SQL expression:

```
select parent_primekey into #R
from Parent
where (select count (*)
       from Child
       where Parent.parent_primekey = Child.foreignkey
       and (condition_expr) ) operation int
```

Group-select operation always returns a one column table. The single attribute is the primary-key of the Parent (equal to the foreign key of the child). Group-select is always associated with a 1:n (or parent-child relationship).

4. **Intersect statement**

   An intersection is always performed between two single-column relations R1 and R2 and translates as follows.

   ERA:

```
R = R1  intersect(join_attribute)  R2
```

   Translates to:

   SQL:

```
select R1.* into #R
from R1, R2
where R1.join_attribute = R2.join_attribute
```

5. **Union statement**

   A union is always performed between two single-column relations R1 and R2 and translates as follows.

   ERA:

   ```
   R = R1 union R2
   ```

   Translates to:

   SQL:

   ```
   select * into #R from R1
   union
   select * from R2
   ```

6. **Subgroup-select statement**

   ERA:

   ```
   R = subgroup-select
       (Child (quantifier_pc (foreignkey (child_condition))
                                          (condition_expr)))
   ```

   Subgroup-select operation can be easily decomposed into two ERA operations (a select and a group-select), and this is how the implementation was done:

   ```
   SR = select (Child (child_condition))
   R = group-select (SR (quantifier_pc (foreignkey
                                          (condition_expr)))
   ```

As it was pointed out in Chapter 5 the type of quantifier, that is, universal type or existential type, makes a difference in the equivalent ERA routine.

To illustrate the difference consider two examples with the quantifier **for all** (universal type):

Example 1: child with condition: "for all Boeing Aircraft"

Give the airline code and headquarters for each airline with more than 10.000 employees all of whose Boeing aircraft have at least two of the parts on board of type P7 with status not ON.

COOL:

```
     select AL#, hqadd from Airline
     where emp_num > 10000
     and for all Airline's (fabricant = "Boeing") Aircraft
       (for at least 2 Aircraft's Parts_on_Board
                (PT# = "P7" and status <> "ON"));
```

ERA:

```
R0 = group_select (Parts_on_Board
       (for at least 2 AC# (PT# = "P7" and status <> "ON")))
R1 = R0 (AC#) pjoin (p0) Aircraft (AC#)
R2 = group_select
          (R1 (for all AL# ((fabricant = "Boeing") AND p0)))
R3 = select ( Airline ( emp_num > 10000))
```

```
R4 = project ( R3 ( AL#))

R5 = R2 (AL#) join Airline (AL#)

R6 = project ( R5 ( AL#))

R7 = R6 intersect(AL#) R4

R8 = R7 (AL#) join Airline (AL#)

R9 = project ( R8 (AL#,hqadd))
```

Example 2: child without condition: " for all aircraft that are of type Boeing and ..."

Give the airline code and headquarters for each airline with more than 10.000 employees all of whose aircraft are (a) of make Boeing and (b) have at least two of their parts-on-board of type P7 with status not ON.

COOL:

```
select AL#, hqadd from Airline

where emp_num > 10000

and for all Airline's Aircraft (fabricant = "Boeing" and

  for at least 2 Aircraft's Parts_on_Board

                    (PT# = "P7" and status <> "ON"));
```

ERA:

```
R0 = group_select (Parts_on_Board

        (for at least 2 AC# (PT# = "P7" and status <> "ON")))

R1 = R0 (AC#) pjoin (p0) Aircraft (AC#)
```

```
R2 = group_select
              (R1 (for all AL# ((fabricant = "Boeing") AND p0)))
R3 = select ( Airline ( emp_num > 10000))
R4 = project ( R3 ( AL#))
R5 = R2 (AL#) join Airline (AL#)
R6 = project ( R5 ( AL#))
R7 = R6 intersect(AL#) R4
R8 = R7 (AL#) join Airline (AL#)
R9 = project ( R8 (AL#,hqadd))
```

The result is different as it was expected. The universal type quantifiers require that all of a group of tuples must be taken into consideration, so the result of the evaluation depends on the whole set (which is unknown). If, instead of **for all**, we had used an existential-type quantifier (e.g.: for more than 10), the two cases would be the same and the corresponding COOL expressions will give identical results.

7. **Projection statement**

   In general the project statement translates as follows.

   ERA:

   ```
   R = project (R1 (selection))
   ```

   SQL:

   ```
   select selection into #R
   ```

```
    from R1
```

where selection is a set of attributes of the relation R.

As discussed in the subsection on the ERA join statement at the beginning of section 6.5.10, in the case of a query with composite parent-to-child genitive relations, the relation resulting from a 'project statement' needs a name change for the column name that is a join attribute in a subsequent join statement. For this special case the 'project statement' translates as follows.

ERA:

```
    R = project (R1 (attribute_1, attribute_2))
```

SQL:

```
    create table #RR
    ( RR type,
      attribute_2 type
    )
    insert #RR select attribute_1, attribute_2 from R1
```

The type of the attribute_1 and attribute_2 can be read from the Sybase system tables because the attributes belong always to tables from the created database.

8. **Pjoin (Possibility Join) statement.** To implement pjoin I have used the outer join operation, since this is very close to pjoin. Outer Join it is implemented in Sybase Transact-SQL and SQL 2.

In a natural join only the rows with matching values in the columns specified in the join condition are included in the results. Sometimes, as in the case of pjoin, it is desirable to retain the non-matching rows as well in the result of a join. Sybase SQL [Syb91b] supports the outer join by providing the specific join operators:

- (a) *= This means include in the result all the rows from the first table, not just the ones where the join columns match. In the result the non matching rows will have a NULL in the columns belonging to the second table.
- (b) =* This means include in the result all the rows from the second table, not just the ones where the join columns match. In the result the non matching rows will have a NULL in the columns belonging to the first table.

ERA:

```
R = R1 (attribute) pjoin (variable) R2 (attribute)
```

SQL:

```
  create table #variable
   (var char(length)/int null)


insert #variable select * from R1
```

```
select * into #R
from R2, #variable
where R2.attribute *= #variable.variable
```

The table #variable has only one column with the name of the pjoin variable and the type of the join field in the pjoin expression. Creation of a new relation is necessary because of the Sybase SQL implementation. When making the outer join (or any join) between relation R1 and R2 with two identical join field names, the result must include duplicate columns with the same join field name, that is, an equi-join is performed. But, as we discussed earlier, if we want to capture the resulting relation in a temporary relation, we get an error from the Sybase SQL server stating that "Column names in each table must be unique" (which makes sense for a relation). So to avoid this, we copy the join field of relation R1 into the relation #variable, but using a different full name.

To select fields that are (or are not) NULL you put the condition "where column_name is [not] NULL". In order to be able to do that and not get an error message, the column should be defined NULL in the CREATE TABLE statement [Syb91b] as has shown above.

6.5.11 Implementation Restrictions and Conventions

The OID is always the first column in the object class table. The primary key is the second column, and the foreign keys are following the primary key. Reflist names are unique for the database.

Chapter 7

System Verification and User View of the System

7.1 Introduction

This chapter deals with two topics: verification that the system works correctly, and a description of how the system is to be used by a novice user.

7.2 System Verification

In general system verification is an important part in a system's development. Verification first needs to be thoroughly performed by the system developer. Further verification then needs to be carried out by independent analysts. The test examples must verify all critical situations implemented by the application.

7.2.1 Methods of Verification

The COOL language examples included in the thesis have equivalent ERA routines and SQL expressions generated by the system itself. The results of these COOL queries have been checked using three methods. One method was logical testing, where both the logic of ERA routines and the equivalent SQL set of expressions were checked for correctness. The second method involved using sample data loaded in an example database and executing the COOL queries against it, as well as checking the result for correctness. The third method involved constructing an equivalent SQL expression for the tested query, executing it, and checking that it retrieved the same data as the COOL expression.

In addition, for a final verification of the system three sample queries, chosen spontaneous by Prof. J. Bradley, were used.

7.3 Sample Verification Queries and Results

For each query there will be shown the equivalent expressions in COOL, ERA and SQL. In the Appendix D.4 it is shown the script file with the executions of the three queries using a set of test data for the 'aircraft maintenance database'.

Query 1 is testing a two level nested quantifier expression and a 'child with condition' involved in a parent-child (or 1:n relationship) genitive relation. Natural quantifiers 'for most' and 'for all' are also tested.

Query 1

List airline code and headquarters location for airlines where most aircraft of type Boeing have all parts on board with status OK.

COOL:

```
select AL#, hqadd from Airline
where for most Airline's (fabricant = "Boeing") Aircraft
        (for all Aircraft's Parts_on_Board (status = "ok"));
```

ERA:

```
R0 = group_select (Parts_on_Board (for all AC# (status = "ok")))
R1 = R0 (AC#) pjoin (p0) Aircraft (AC#)
R2 = subgroup_select
```

```
               (R1 (for most (AL# (fabricant = "Boeing")) (p0)))
R3 = R2 (AL#) join Airline (AL#)
R4 = project ( R3 (AL#,hqadd))
```

SQL:

```
select AC# into #R0
from Aircraft
where not exists ( select *
                   from Parts_on_Board
                   where Aircraft.AC# = Parts_on_Board.AC#
                   and not (status = "ok") )


create table #p0
( p0 char(4) null )


insert #p0 select * from #R0


select * into #R1
from Aircraft, #p0
where Aircraft.AC# *= #p0.p0


select * into #SR1
from #R1
where fabricant = "Boeing"
```

```
select AL# into #R2

from Airline

where  ( select count (*)

         from #SR1

         where Airline.AL# = #SR1.AL#

         and p0 is not null )

         >

       ( select count (*)

         from #SR1

         where Airline.AL# = #SR1.AL#

         and not (p0 is not null))


select Airline.* into #R3

from #R2, Airline

where #R2.AL# = Airline.AL#


select AL#,hqadd into #R4

from #R3


select distinct * from #R4


go
```

Using sample data for the Aircraft Maintenance database (see Appendix D.2) the expected result for query 1 was AL# = "AL" and hqadd = "Rome". This result is confirmed by the output of Sybase isql (see Appendix D.4).

Manual **SQL**

```
select AL#, hqadd from Airline
where (select count (*)
       from Aircraft
       where fabricant = "Boeing"
       and Aircraft.AL# = Airline.AL#
       and AC# not in (select AC# from Parts_on_Board
                       where status <> "ok")
       )
       >
      (select count (*)
       from Aircraft
       where fabricant = "Boeing"
       and Aircraft.AL# = Airline.AL#
       and AC# in (select AC# from Parts_on_Board
                   where status <> "ok")
       )
```

This retrieval gives the same result as the SQL set of expressions generated by the COOL system, that is, AL# = "AL" and hqadd = "Rome", when it is executed with the same data. This result is confirmed by the output of Sybase isql (see Appendix

D.4).

Query 2 is testing a n:m relationship, specified and implemented as a composite parent-child and child-parent genitive relation. Natural quantifiers 'for at least 2', 'for at least 1' and 'for its' are also tested.

Query 2.

Give the PT# and quantity for each type of part in inventory that has (a) at least 2 shipments from suppliers in Los Angeles, and (b) has status 'defect' on at least one aircraft on which it is used.

COOL:

```
select PT#, qty from PartType_Inventory
where for at least 2 PartType_Inventory's Shipment_Data
  (for its Shipment_Data's Supplier (address = "Los Angeles"))
and for at least 1 PartType_Inventory's Parts_on_Board
                                               (status = "de");
```

ERA:

```
R0 = select (Supplier (address = "Los Angeles"))
R1 = project (R0 (S#))
R2 = R1 (S#) join Shipment_Data (S#)
R3 = project (R2 (SH#))
R4 = R3 (SH#) pjoin (p0) Shipment_Data (SH#)
R5 = group_select (R4 (for at least 2 PT# (p0)))
R6 = group_select
         (Parts_on_Board (for at least 1 PT# (status = "de")))
```

```
R7 = R5 intersect(PT#) R6

R8 = R7 (PT#) join PartType_Inventory (PT#)

R9 = project ( R8 (PT#,qty))
```

SQL:

```
select * into #R0

from Supplier

where address = "Los Angeles"


select S# into #R1

from #R0


select Shipment_Data.* into #R2

from #R1, Shipment_Data

where #R1.S# = Shipment_Data.S#


select SH# into #R3

from #R2


create table #p0

( p0 char(4) null )


insert #p0 select * from #R3
```

```
select * into #R4
from Shipment_Data, #p0
where Shipment_Data.SH# *= #p0.p0

select PT# into #R5
from PartType_Inventory
where ( select count (*)
        from #R4
        where PartType_Inventory.PT# = #R4.PT#
        and p0 is not null ) >= 2

select PT# into #R6
from PartType_Inventory
where ( select count (*)
        from Parts_on_Board
        where PartType_Inventory.PT# = Parts_on_Board.PT#
        and status = "de" ) >= 1

select #R5.* into #R7
from #R5, #R6
where #R5.PT# = #R6.PT#

select PartType_Inventory.* into #R8
from #R7, PartType_Inventory
```

```
where #R7.PT# = PartType_Inventory.PT#


select PT#,qty into #R9

from #R8


select distinct * from #R9


go
```

Using sample data for the Aircraft Maintenance database the expected result for query 2 was PT# = "P11" and qty = 1000. This result is confirmed by the output of Sybase isql (see Appendix D.4).

Manual **SQL**

```
select PT#, qty from PartType_Inventory
where (select count (*)
       from Shipment_Data
       where PartType_Inventory.PT# = Shipment_Data.PT#
       and S# in (select S# from Supplier
                  where address = "Los Angeles"))>=2
and (select count (*)
     from Parts_on_Board
     where PartType_Inventory.PT# = Parts_on_Board.PT#
     and status = "de")>=1
```

This retrieval gives the same result as the SQL set of expressions generated by the COOL system, that is, PT# = "P11" and qty = 1000, when it is executed with the same data. This result is confirmed by the output of Sybase isql (see Appendix D.4).

Query 3 is testing a composite parent-child (or 1:n) genitive relation.

Query 3.

Which maintenance depot in Montreal has carried out at least two service projects in each of which all jobs involved part type 'P4'.

COOL:

```
select MD# from Maintenance_Depot

where address = "Montreal"

and for at least 2 Maintenance_Depot's Service's Service_Project

        (for all Service_Project's Job (PT# = "P4"));
```

ERA:

```
R0 = group_select (Job (for all SVP# (PT# = "P6")))

R1 = R0 (SVP#) pjoin (p0) Service_Project (SVP#)

RR2 = project (Service (SV#, MD#))

R3 = RR2 (SV#) join R1 (SV#)

R4 = group_select (R3 (for at least 2 MD# (p0)))

R5 = select ( Maintenance_Depot ( address = "Montreal"))

R6 = project ( R5 ( MD#))

R7 = R4 (MD#) join Maintenance_Depot (MD#)

R8 = project ( R7 ( MD#))
```

```
R9 = R8 intersect(MD#) R6

R10 = R9 (MD#) join Maintenance_Depot (MD#)

R11 = project ( R10 (MD#))
```

SQL:

```
select SVP# into #R0

from Service_Project

where not exists ( select *

                   from Job

                   where Service_Project.SVP# = Job.SVP#

                   and not (PT# = "P6") )


create table #p0

( p0 char(4) null )


insert #p0 select * from #R0


select * into #R1

from Service_Project, #p0

where Service_Project.SVP# *= #p0.p0


create table #RR2

( RR2 char(4),

  MD# char(4))
```

```
insert #RR2 select SV#, MD# from Service


select #RR2.*, #R1.* into #R3

from #RR2, #R1

where #RR2.RR2 = #R1.SV#


select MD# into #R4

from Maintenance_Depot

where ( select count (*)

        from #R3

        where Maintenance_Depot.MD# = #R3.MD#

        and p0 is not null ) >= 2


select * into #R5

from Maintenance_Depot

where address = "Montreal"


select MD# into #R6

from #R5


select Maintenance_Depot.* into #R7

from #R4, Maintenance_Depot

where #R4.MD# = Maintenance_Depot.MD#
```

```
select MD# into #R8
from #R7

select #R8.* into #R9
from #R8, #R6
where #R8.MD# = #R6.MD#

select Maintenance_Depot.* into #R10
from #R9, Maintenance_Depot
where #R9.MD# = Maintenance_Depot.MD#

select MD# into #R11
from #R10

select distinct * from #R11

go
```

Using sample data for the Aircraft Maintenance database the expected result for query 3 was MD# = "M1" . This result is confirmed by the output of Sybase isql (see Appendix D.4).

Manual **SQL**

```
select MD# from Maintenance_Depot
```

```
where address ="Montreal"
and (select count (*)
     from Service, Service_Project
     where Service.MD# = Maintenance_Depot.MD#
     and Service.SV# = Service_Project.SV#
     and SVP# not in (select SVP# from Job
                      where PT# <> "P4"))>=2
```

This retrieval gives the same result as the SQL set of expressions generated by the COOL system, that is, MD# = "M1", when it is executed with the same data. This result is confirmed by the output of Sybase isql (see Appendix D.4).

7.4 User View of the System

In general, a user of the prototype implementation of COOL has to define the object schema of the database he is working with, to load it, query it, and update it.

Prior to any work with the database, any user that wants to use the prototype front-end has to have set in the '.login' file, the Sybase environment as follows:

```
setenv DSQUERY sybase_fsb
setenv SYBASE /usr/local/sybase
```

, and include in the 'set path' command, Sybase path '/usr/local/sybase/bin'.

Every command to the COOL system is given in a text file with the extension '.cool'. The user command for executing the COOL source file is:

```
     coo filename.cool
```

Any execution of a 'coo' command will generate a translation of the COOL statements to an Extended Relational Algebra routine, in the file 'filename.era' and a translation in SQL for Sybase, in the file 'filename.sql'. If the COOL commands represent a create object class, or any of insert, update or delete instance of an object class, the output file of the 'coo' command will be only the SQL file. The other file will be empty. If the COOL commands represent a query, there will be two output files, the '.era' file and the '.sql' file as well.

Basically a **coo filename.cool** command will do the following:

- Will delete 'filename.era' and 'filename.sql' files from previous executions.
- Will execute the command 'cool filename.cool' that will translate COOL into ERA and generate the output into 'filename.era'.
- Will execute the command 'cool filename.era' that will translate COOL into ERA and generate the output into 'filename.sql'.
- Will send the file 'filename.sql' to the Sybase interface 'isql' for the SQL execution.

After giving a *coo filename.cool* command, its result will come out as Sybase interface 'isql' gives it. In the Appendix D of the thesis we will show some script files of executions with the prototype front-end.

In the **coo command** the extension '.cool' is optional, but **the COOL source file has to have the extension '.cool'**.

Any COOL statement ends in ';'.

7.4.1 Database Definition

Suppose we use the Aircraft Maintenance database as an example.

We define every object class in the database using the 'create object class' statement (see 'CREATE statement' in Chapter 6), as we do in the following example for the class 'Airline':

```
create object class Airline
( AL# CHAR(2),
hqadd    char(30),
emp_num  int,
primary key AL#);
```

The COOL definition for the entire Aircraft Maintenance database from figure 6.4 is included in the Appendix C.1. The 'create' statements for all the classes in the database can be included in a single file with the extension '.cool' , for example 'create.cool'.

7.4.2 Database Loading

To insert data into the object classes we can do it only instance by instance in COOL (see 'INSERT statement' in Chapter 6). When we have parent child relationships between the entities of the database we need to load first the parent instances and then the child instances. If we do not do it this way, the system will prevent us by inserting the 'orphan' child instances with the error message:

```
Class 'parent_name' has no instance with primary key = 'value'.
You try to insert a child without a parent.
```

For this reason it is desirable to load first the parents and then the children. We can have all the insert statements for all the database in the same '.cool' file or we can have an insert file for each class in the database. For example the insert file, 'insertAirline.cool' for the class Airline is as follows.

```
insert obj ins into Airline
  ( AL# : "RO", hqadd : "Bucharest", emp_num : 3000);
insert obj ins into Airline
  ( AL# : "AL", hqadd : "Rome", emp_num : 6000);
insert obj ins into Airline
  ( AL# : "AF", hqadd : "Paris", emp_num : 7000);
insert obj ins into Airline
  ( AL# : "BA", hqadd : "London", emp_num : 10000);
```

The 'create.cool' file and the the insert files, insertAirline.cool, insertAircraft.cool, and so on are created with a text editor and then executed together with a shell script file. The shell script file 'loadAMD' that will create and load the example database, will contain the following statements:

```
#!/bin/sh
# statements for cleaning and initializing COOL system tables
isql -Urata -Pcarmen < clean.sql
isql -Urata -Pcarmen < init.sql
# create the database
coo create.cool
echo The database was created
```

```
# load the database

coo insertAirline.cool

coo insertAirline.cool

echo Airline class loaded

coo insertMD.cool

echo Maintenance_Depot class loaded

coo insertPTI.cool

echo PartType_Inventory class loaded

coo insertSup.cool

echo Supplier class loaded

coo insertAircraft.cool

echo Aircraft class loaded

coo insertT.cool

echo Technician class loaded

coo insertServ.cool

echo Service class loaded

coo insertServP.cool

echo Service_Project class loaded

coo insertJob.cool

echo Job class loaded

coo insertShipD.cool

echo Shipment_Data class loaded

coo insertPB.cool

echo Parts_on_Board class loaded
echo The database was created and loaded successfully!
```

Creating and loading the Aircraft Maintenance database will be executed in Appendix D.1.

7.4.3 Submission of COOL queries

After the database is created and loaded the user can submit queries. A query is written in a file with the extension '.cool' and then submitted to the database with the command:

```
coo query.cool
```

The output will come out in a Sybase isql format. The contents of the script file 'coo' is given in Appendix D.8.

7.4.4 Database update

We can submit 'update' and 'delete instance' statements in the same way in which we submit queries to the COOL front-end. In a '.cool' file we can input an update instance statement, such as:

```
update Airline where AL# = "RO" set hqadd : "Arad";
```

In the same way in a '.cool' file we can input a delete instance statement, such as:

```
delete from Airline where AL# = "BA";
```

Shell script files with examples of insert, update and delete statements are included in Appendix D.3, D.5 and D.6. A shell script file with typical error messages is also included in Appendix D.7.

Chapter 8

Conclusions and Future Work

8.1 Future COOL design and implementation work

Future work on COOL could include the following:

- An optimizer for ERA routines.
- Second possible DDL.
- Implementation of inheritance.
- Design and implementation of Complex Objects: including the definition of, storage of, and update of Complex Objects, and interface of Complex Objects with programming languages.
- Design and implementation of functions, including the definition and inheritance of functions.

Some of these issues have been partially investigated, as discussed below.

8.1.1 Second possible DDL

A second approach to the CREATE command would be to declare the object classes without the relationships and declare the relationships separately. This approach embodies a higher level of abstraction. It implements the structural object orientation of the semantic data models, such as the Entity Relationship model, in the data definition language. In this way, when designing the database schema we do not need to map the conceptual level described by a semantic data model to an extended nested relational model in order to write the data definition. Thus, we eliminate an intermediate step in the conceptual design of a database.

The syntax in this case would be:

```
/*  create a object class  */
<create_objectcls> ::=
        CREATE OBJ[ECT] CL[AS]S <objectcls>
                        '('<objectcls_element_commalist> ')'
<objectcls_element_commalist> ::=
              <objectcls_element>
           |  <objectcls_element_commalist> ',' <objectcls_element>
<objectcls_element> ::=
              <attribute_def>
           |  [ <objectcls_key_def> ]
<attribute_def> ::=
                <attribute> <attribute_type>
<objectcls_key_def> ::=
               CANDIDATE KEY '(' <attribute_commalist> ')'
           |   PRIMARY KEY <attribute>
/*   create a relationship   */
<create_rel> ::=
            CREATE REL 1toM '(' <relationship_field_commalist> ')'
```

```
          | CREATE REL MtoM '(' <relationship_field_commalist> ')'
          | CREATE REL ISA '(' <relationship_field_commalist_inh> ')'
<relationship_field_commalist> ::=
            <relationship_field>
          | <relationship_field_commalist> ',' <relationship_field>
<relationship_field> ::=
            <objectcls_1> [: <reference_attribute_1>] <objectcls_2>
                          [: <reference_attribute_2>]
<relationship_field_commalist_inh> ::=
            <relationship_field_inh>
          | <relationship_field_commalist> ','
                                         <relationship_field_inh>
<relationship_field_inh> ::=
             <subclass> [: <attribute>] INHERITS FROM <superclass>
                        [: <attribute>]
```

The meanings of the identifiers <objectcls_1>, <reference_attribute_1>, <objectcls_2>, <reference_attribute_2> for 1:n relationships, that is, for the command CREATE REL 1toM, and for n:m relationships, that is, CREATE REL MtoM, are as follows:

	1toM	MtoM
objectcls_1	parent	objectclsX
objectcls_2	child	objectclsY
reference_attribute_1	reflist	reflist_Y

```
reference_attribute_2             superkey          reflist_X
```

Identifiers <reference_attribute_1> and <reference_attribute_2> are optional. When they are omitted only one relationship is assumed between <objectcls_1> and <objectcls_2> and the name of the relationship (described by the reference list) is generated by the system.

Some examples with the second approach to Create are:

```
create object class Airline
( AL#   CHAR(2),
hqadd    char(30),
emp_num  int,
primary key AL#
);
create obj cl Aircraft
(AC#    char(4),
fabricant char(20),
type    char(4),
primary key AC#
);
create rel 1toM ( Airline Aircraft );
```

In the example above, the reference attributes are generated by the system. Reference_attribute_1 will be *Airclist_Air* and reference_attribut_2 will be *AL#*.

An example of two distinct 1:n relationships between the related classes Construct and Part_Type_Inventory is the following:

```
create rel 1toM (PartType_Inventory:inner_parts Construct:PT_outer,
                 PartType_Inventory:outer_parts Construct:PT_inner);
```

The <reference_attribute_1> and <reference_attribute_2> for the first relationship are: *inner_parts*, which is a set of OIDs of the PartType_Inventory related instances in the object-class Construct, and *PT_outer*, which is the OID of the Construct related parent instance in the object-class PartType_Inventory. The reference_attributes for the second relationship are: *outer_parts* and *PT_inner*. The two distinct reference attributes of the second example with 'create rel' need to be specified by the user in order to have helpful, suggestive names.

The advantage of this second data definition for COOL is the higher level of abstraction, which is similar to the conceptual level defined by the the semantic models, such as the Entity Relationship model. Using this higher level of abstraction the user does not have to be concerned about using foreign keys to define relationships. From the definition of relationships, the 'create rel' statements, the system will make all the necessary links required by a relationship implementation. Thus, this true conceptual database definition takes lower level detail from users' tasks and transfers it to the DBMS.

8.1.2 Inheritance in COOL

Inheritance in COOL is essentially about how to handle IS-A 1:1 relationships.

Alternatively, inheritance is a reusability mechanism that makes possible for a class called **subclass** to be defined on the basis of the definition of an ISA 1:1 related class called a **superclass**.

Consider the following ISA class hierarchy: Aircraft, Helicopter, FW_Aircraft, where aircraft are either helicopters or fixed wing. We include in the hierarchy a class Company that is in 1:n relationship with class Aircraft, that is, a company has many aircraft. We also include an object class Service, where an aircraft requires many services. The COOL data definition of the class hierarchy is given below:

```
Company: create obj class Company
           ( co#      char(4),
             hqadd    char(30),
             emp_num  int,
             primary key co#
           )
Aircraft: create obj class Aircraft
            ( ac#    char(4),
              co#    char(4),
              manufacturer char(20),
              type     char(4),
              primary key ac#,
              super key (c# (Company:airclist))
            )
Helicopter: create obj class Helicopter
              ( h#      char(4),
                ac#     char(4),
                rotor#  int,
                primary key h#,
                super key (ac# (Aircraft))
                inherits (ac# (Aircraft)
              )
Fixed_Wing_Aircraft: create obj class FW_Aircraft
                       ( fw# char(4),
                         ac#     char(4),
                         engine_type char(10),
                         engine#     int,
                         primary key fw#,
                         super key (ac# (Aircraft))
                         inherits (ac# (Aircraft)
                       )
Service:  create obj class Service
            ( s#      char(4),
              ac#     char(4),
              description char(80),
              primary key s#,
              super key (ac# (Aircraft))
            )
```

The classes Aircraft and Helicopter or FW_Aircraft above involve single inheritance and one level superclass - subclass 1:1 relationship.

8.1.3 Single Inheritance and one level Superclass - Subclass Relationship

The ISA relationship defined above gives rise to 6 distinct situations with respect to genitive relations:

1. **Parent** to **child with inheritance** genitive relation, for example: *Company's Helicopters*

2. **Child with inheritance** to **parent** genitive relation, for example: *Helicopter's Company*

3. **Parent with inheritance** to **child** genitive relation, for example: *FW_Aircraft's Service*

4. **Child** to **parent with inheritance** genitive relation, for example: *Service's FW_Aircraft*

5. **Parent with inheritance** to **child with inheritance** genitive relation, for example: *Special_Service's FW_Aircraft*

6. **Child with inheritance** to **parent with inheritance**, genitive relation, for example: *FW_Aircraft's Special_Service*

For each of the situations above I will give query examples and I will show how COOL expressions can be reduced to an ERA routine.

1. Query example involving the **parent - child with inheritance** genitive relation.

   Query: *Get the address of a maintenance company that services only helicopters with 4 rotors.*

COOL:

```
select HQ from Company
where for all Company's Helicopters (rotor# = 4)
```

ERA:

```
R0 = project (Aircraft (primekey,  superkey))
R1 = R0 (Aircraft_primekey) join Helicopter (superkey)
R2 = group_select (R1 (for all Aircraft_superkey (rotor# = 4)))
R3 = Company (primekey) join R2 (Aircraft_superkey)
R4 = project (R3 (HQ))
```

2. Query example involving the **child with inheritance - parent** genitive relation.

   Query: *Get all the details of each helicopter maintained by companies located in Los Angeles.*

   COOL:

```
select * from Helicopter
where for its Helicopter's Company ( HQ = 'LA')
```

ERA:

```
R0 = project (Aircraft (primekey, superkey))
R1 = R0 (Aircraft_primekey) join Helicopter (superkey)
```

```
R2 = select (Company (HQ = 'LA'))

R3 = R1 (Aircraft_superkey) join R2 (Company_primekey)

R4 = project (R3 (Helicopter_primekey))

R5 = R4(Helicopter_primekey) join Helicopter (primekey)

R6 = project (R5 (*))
```

3. Query example involving the **parent with inheritance - child** genitive relation.

   Query: *Get the number of engines on each fixed wing aircraft for which the majority of performed maintenance services were computer repair.*

   COOL:

   ```
   select engines# from FW_Aircraft
   where for most FW_Aircraft's Service
                  (description = ''computer repair'');
   ```

   ERA:

   ```
   R0 = group_select (Service (for most Aircraft_primekey
                              description = 'computer repair')))
   R1 = FW_Aircraft (superkey) join R0 (Aircraft_primekey)
   R2 = project (R1 (engines#))
   ```

4. Query example involving the **child - parent with inheritance** genitive relation.

Query: *Give the description of each service performed on fixed wing aircraft for which the engine type is turbine.*

COOL:

```
select description from Service
where for all Services's FW_Aircraft (enginetype = 'turbine')
```

ERA:

```
R0 = project (Aircraft (primekey, superkey))
R1 = R0 (Aircraft_primekey) join FW_Aircraft (superkey)
R2 = select ( R1 (enginetype = 'turbine'))
R3 = Service (superkey) join R2 (Aircraft_primekey)
R4 = select (R3 (description))
```

5. Query example involving the **child with inheritance - parent with inheritance** genitive relation.

   Now suppose additionally that Services also have an ISA 1:1 relationship with Regular Services and Special Services.

   Query: *Give the description of Special Services for aircraft with two engines.*

   COOL:

```
select sp_description from Special_Service
where for its Special_Service's FW_Aircraft (engines#=2);
```

   ERA:

```
R0 = project (Aircraft (primekey))

R1 = R0 (Aircraft_primekey) join FW_Aircraft (superkey)

R2 = select (R1 (engines# = 2))

R3 = project (R2 (Aircraft_primekey))

R4 = project (Services (primekey, superkey))

R5 = R4 (Services_primekey) join Special_Services (superkey)

R6 = R3 (Aircraft_primekey) join R5 (Services_superkey)

R7 = project (R6 (sp_description))
```

6. Query example involving the **parent with inheritance - child with inheritance** genitive relation.

   Query: *Get the number of engines on fixed wing aircraft that all need computer repair.*

   COOL:

```
select engines# from FW_Aircraft

where for all FW_Aircraft's Special_Service

                  (description = ``computer repair'');
```

   ERA:

```
R0 = project (Services (primekey, superkey))

R1 = R0 (Services_primekey) join Special_Services (superkey)

R2 = group-select (R1 (for all Services_superkey

                          (description = ``computer repair'')))
```

```
    R3 = project (Aircraft (Aircraft_primekey))
    R4 = R3 (Aircraft_primekey) join FW_Aircraft (superkey)
    R5 = R4 (Aircraft_primekey) join R2 (Services_superkey)
    R6 = project (R5 (engines#))
```

8.1.4 Single Inheritance and multiple level Superclass - Subclass relationship

For the case of single inheritance and multiple level superclass - subclass relationship, we have the same cases as above. Now suppose a Helicopter can be either a Transport (Transport_Hely) or a Passenger Helicopter (Passenger_Hely). For the first category of Cool Query 'parent- child with inheritance genitive relation', the equivalent COOL expression and ERA routine, are:

Query: *Get the address of each company that uses transport helicopters with maximum loading of 50 tons.*

COOL:

```
select hqadd from Company
where for all Company's Transport_Hely (maxload = 50)
```

ERA:

```
R0 = project (Aircraft (primekey,  superkey))
R1 = R0 (Aircraft_primekey) join Helicopter (superkey)
R2 = project (R1 (Helicopter_primekey, Aircraft_primekey))
R3 = Transport_Hely (superkey) join R2 (Helicopter_primekey)
R4 = group_select (R3 (for all Aircraft_superkey (maxload = 50)))
```

```
R5 = Company (primekey) join R4 (Aircraft_superkey)

R6 = project (R5 (hqadd))
```

8.1.5 Composite or complex objects

A composite object can be defined as an aggregation of objects [Cat91] (e.g.: electronic components can be grouped together to form a computer, chapters may be grouped together to form a document). Composite objects have also a hierarchical structure (e.g.: paragraphs are parts of sections, sections are parts of chapters and chapters are parts of documents).

The research described in this thesis has not included composite object handling in COOL. Investigation of composite objects, models, languages and implementation systems would require a lengthy research project. The goal of a database system that allows composite objects has given rise to many proposals for models, beginning with the nested relations model.

To give the reader a general idea of what is involved, suppose we have an object type hierarchical database structure: object class S is parent of object classes A, B and C; object class A is parent of object classes X, Y, Z and object class B is the parent of object classes U, V, W.

Considering S1, S2,... instances of object class S; A1, A2,... instances of object class A and so on, the hierarchical structure of a composite object can be laid out as in figure 8.1 [Bra93b].

As it was shown in Chapter 5, language constructs for complex objects are available in COOL. This involves COOL expressions for retrieval, concentration and creation (complex object views) of composite objects. Methods for reduction of such

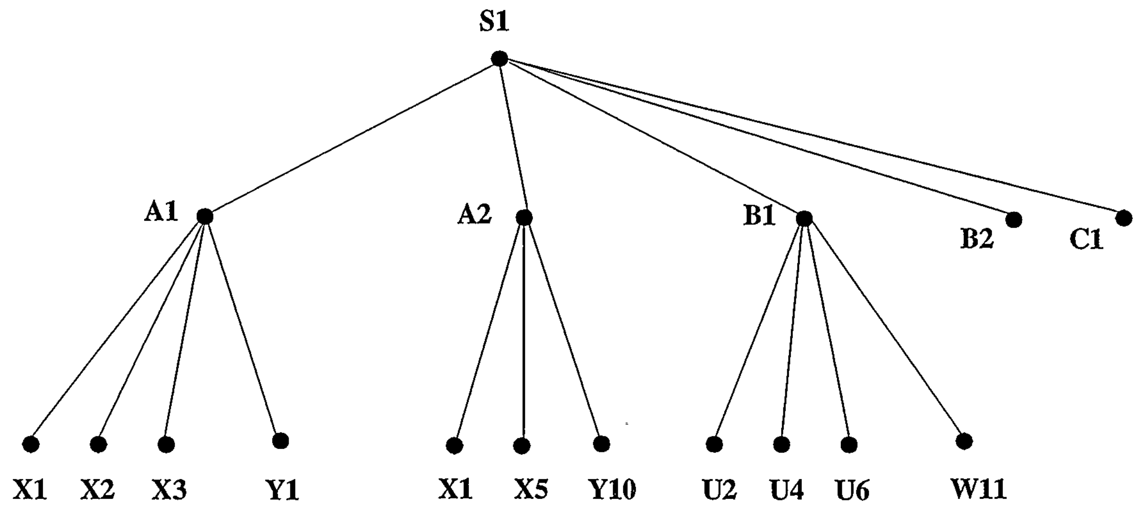

Figure 8.1: Composite Object

composite object COOL expressions have still to be investigated.

A possible way of implementing composite objects is by using relations. In this approach the objects used for constructing composite objects would be stored in a relational database as relations. COOL could then generate a composite object of the type shown in figure 8.1 from the underlying relations.

The composite object could be stored in a relation as a set of all the pairs (parent_instance OID, child_instance OID). The root of the composite object hierarchy could be a tuple with the parent NULL. A leaf in the three could be a tuple with the child NULL.

A host-program with embedded COOL commands would be needed to retrieve a composite object from the database in order to transfer the composite object instances one by one to a host program structure variable.

A major problem is updating a composite object.

8.1.6 Functions or Methods

The novel concept that is central to OO approach is undoubtingly the function (called also procedure or method). With functions a new layer has been added to the already well known structural OO, namely behavioral OO.

Functions add to database languages an important missing feature. This feature is the computational completeness that we find in programming languages.

A variety of approaches has been taken to support functions in OO database systems. But a precise framework for functions in COOL has not yet been designed. As a consequence functions were not implemented in the system described in this thesis. The present section merely establishes a framework for future functions and ADT design in COOL, since COOL can be extended to use user-defined functions (methods, ADT or Abstract Data Types) and to perform encapsulation of data with methods.

Functions could be written either in COOL (a COOL function), if it is possible or in a host programming language (an external function). This idea for creating functions was used in the draft SQL3 standard [CMCG94] and can be applied to COOL as well.

COOL functions present more advantages than external host programming language functions because they can be optimized, because the switching to a host-language context can be avoided, and because user-defined types can be passed as arguments.

We could also improve the ease of writing COOL functions by allowing COOL to be used in a procedural manner. Some commercial database products permit users to invoke stored procedures written in a language native to the database product. Sybase has a language of this kind called "Control-of-Flow". This language provides special statements like: if...else, while, begin...end, goto label, return, and so on and permits a procedural execution of the SQL statements. Incorporating such procedural language constructs into COOL would facilitate the writing of user-defined functions.

In the SQL3 draft standard [CMCG94], the programming language type has not been implemented yet. This means that external functions written in a host programming language (e.g.: C) need an interface to translate the ADT instances passed as arguments to the function into host language types. ADTs can be passed to external functions in the form of language-type instances.

8.2 Summary and Conclusions

In this thesis, we first presented an evolution of the database models, starting with the prerelational models, then the relational model, the milestone in the development of databases, and finishing with the object-oriented models. Further on we presented the concepts of the major trends in the OODBMS development, and the basics of the query languages. In the second part of the thesis, we comprehensively presented a novel declarative object-relational language, COOL, the Extended Relational Algebra (ERA) for COOL, and the first implementation of COOL. The thesis concluded with a user view of the system, system verification and future work.

Almost all of the database products developed over the past 10 years are based on the relational technology. The simplicity of the relational is beneficial, because it results model is a benefit in ease of use and mathematical tractability, but it is also a limitation. There is widespread agreement that conventional normalized relations are cumbersome at best when it comes to dealing with certain nontraditional database applications, such as CAD/CAM, text processing, forms management, and picture and voice processing. From the large variety of systems that were proposed to meet the new database applications demanding we can see clearly two main trends in solving the problem:

1. Extend the relational model appropriately,

2. Replace the relational model entirely by a new model.

Object-oriented systems represent an example of the second solution but the lack of a solid theoretical basis is a major criticism of the object-oriented approach. The first solution can be characterized as an attempt to provide a sound mathematical foundation for the object-oriented approach by extending the relational model to incorporate object-oriented concepts.

Several researchers have suggested that one way to increase the functionality of the relational model is to drop the requirement that relations be normalized. The non-first-normal-form (NFNF or NF2) relation and the concepts of semantic modeling are the basis for the development of the extended relational models that represent the foundation of the Object-Relational Database Systems. Another big advantage of the extended relational approach to OODBMS is the non procedural or declarative query language. The aim of this thesis was the implementation of a declarative query language, object-relational, called COOL (A Composite Object-Oriented Language), which was designed as an extension of SQL for use with an

NF2 extended relational database. COOL is relational because is based on the set-theory and is object-oriented because of its underlying object-oriented model and the object-orientation reflected in the language constructs (e.g.: the genitive relation). In COOL we think in terms of objects only, by contrast to SQL where we think in terms of entire relations. COOL proves to be easy to use (easier than SQL) and has a natural language structure given both by the genitive relation (genitive case in the English language), and the natural quantifiers which are used much as in a natural language. COOL is unique in the field of declarative languages, which makes the implementation of COOL the first of its kind.

This thesis described an approach to implementing a declarative object-relational database language. The prototype implementation of COOL followed the option of a front-end to an existing relational database system rather than implementing a full OODBMS, which would have been a much more complex and far more time-consuming task.

For the translation of COOL, a two step translation was chosen: (1) reduction of COOL expressions to Extended Relational Algebra (ERA) routines, and (2) translation of ERA routines into a set of SQL expressions. The translation of COOL into ERA routines offers flexibility for future development of a full OODBMS, and the translation of ERA routines into SQL expressions gives us portability of the prototype, which can be executed on any relational system that supports SQL. In generating ERA routines I aimed for optimum code, as far as possible, following the basic rules of query processing optimization. Building a query optimizer for ERA routines is a future research task. The prototype was built on top of Sybase DBMS.

I have designed and implemented also the data definition language for COOL, and

I have designed but not implemented the inheritance mechanism based on genitive relations for ISA relationships.

Even if this prototype version does not have all the features of the designed COOL, like complex objects and inheritance, it proves that a language like COOL has valuable properties, such as:

- *Powerful* (more powerful than SQL): concise expressions,
- *Easy to use*: ease of construction of expressions,
- *Reliable*: errors in expressions less likely than with SQL,
- *Flexible*: many different ways of constructing an expression, and
- *Natural*: expression structures like those of natural language.

COOL could be useful for a large variety of applications, especially scientific applications that deal with very complex entities. The COOL approach to declarative database languages also seems ideal for future oral interrogation of databases. This thesis has demonstrated the practical feasibility of the COOL approach.

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non First Normal Form Relations to Represent Hierarchically Organized Data. In *Proc. of the third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 191–200, 1984.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st Int'l. Conf. on Deductive and Object-oriented Databases*, pages 40–57, 1989.

[AH84] S. Abiteboul and R. Hull. IFO: a Formal Semantic Database Model. In *Proc. of ACM-SIGMOD Conference on Principles of Database Systems*, 1984.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Ban93] F. Bancilhon. Object Database Systems: Functional Architecture. In *Object Technologies for Advanced Software. First JSSST International Symposium. Proceedings*, pages 163–75, 1993.

[BBB+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2, an Object-Oriented Database System. In *Advances in Object-Oriented Database Systems. 2nd Intl. Workshop on Object-Oriented Database Systems. Proceedings. LNCS 334*, pages 1–22. Springer-Verlag, 1988.

[BCG+87] J. Banerjee, H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *ACM TOIS*, 5(1):3–26, 1987.

[Bee88] D. Beech. A Foundation for Evolution from Relational to Object Databases. In *Advances in Database Technology - EDBT '88. Intl. Conference on Extending Database technology. Proceedings. LNCS 303*, pages 251–271. Springer-Verlag, 1988.

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5(4):353–382, October 1990.

[Bla93] J.A. Blakeley. ZQL[C++]: Integrating the C++ language and an object query capability. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 138–144, May 1993.

[BLM91] O. Boucelma and J. Le Maitre. An extensible functional Query Language for an Object Oriented Database System. In *Proc. of the Second Int'l Conf. on Deductive and Object-Oriented Databases*, pages 567–581, December 1991.

[BM93] E. Bertino and L. Martino. *Object-Oriented Database Systems.* Addison-Wesley, 1993.

[BMO$^+$89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases, and Applications*, chapter 12, pages 283–308. Addison-Wesley, 1989.

[BNPS92] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: the notion and the issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–37, June 1992.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, October 1991.

[Bra78] J. Bradley. An extended owner-coupled set data model and predicate calculus for data base management. *ACM Transactions on Data Base Systems*, 3(4):385–415, 1978.

[Bra83] J. Bradley. SQL/N and attribute/relation associations implicit in functional dependencies. *Intl. J. Computer & Information Science*, 12(20):65–86, 1983.

[Bra87] J. Bradley. *Introduction to Data Base Management in Business.* HRW The Dryden Press, 1987.

[Bra88] J. Bradley. A group-select operation for relational algebra and implications for data base machines. *IEEE Transactions on Software Systems*, 14(1):126–29, 1988.

[Bra92a] J. Bradley. A Genitive Relational Tuple Calculus for an N^2F Object-Oriented Relational Data Model. Technical report, Department of Computer Science, The University of Calgary, September 1992. 92/488/26.

[Bra92b] J. Bradley. An Object-Relationship Diagrammatic Technique for Object-Oriented Database Definitions. *Journal of Database Administration*, 3(2):1–11, 1992.

[Bra92c] J. Bradley. Genitive Relations and the Composite Object-Oriented Language COOL for Object Support in N^2F Relational Data Bases. Technical report, Department of Computer Science, The University of Calgary, Aug 1992. 92/482/20.

[Bra92d] J. Bradley. Recursive Relationships and Natural Quantifier Set Theoretic Expression Techniques. *Computer Journal*, 35(4):A343–8, August 1992.

[Bra93a] J. Bradley. COOL: A Composite Object Oriented Language for N^2F Object-Oriented Relational Data Bases. Technical report, Department of Computer Science, The University of Calgary, March 1993. 93/512/17.

[Bra93b] J. Bradley. COOL Concepts and Semantics for Definition, Concetration and Manipulation of Composite Objects in an N^2F Relational data base. Technical report, Department of Computer Science, The University of Calgary, March 1993. 93/513/18.

[Bra94] J. Bradley. Extended Relational Algebra for Reduction of Natural Quantifier COOL Expressions. Technical report, Department of Computer Science, The University of Calgary, May 1994. 94/540/09.

[Bro91] A.W. Brown. *Object-Oriented Databases and their Applications in Software Engineering.* McGraw-Hill, 1991.

[Cat91] R.G.G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[Cat93] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.

[CC89] Q. Chen and W. Chu. A high-order logic programming language (HILOG) for non-1NF databases. In *Proc. of the First International Conference on Deductive and Object-Oriented Databases*, pages 396–418, December 1989.

[CDG+89] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekit, and S.L. Vandenberg. The EXODUS Extensible

DBMS Project: An Overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan-Kaufmann, 1989.

[CDKK85] H. Chou, D. DeWitt, R. Katz, and A. Klug. Design an Implementation of the Wisconsin Storage System. *Software Practice and Experience*, 15(10), October 1985.

[Cha78] C.L. Chang. DEDUCE-2: Further investigations of deduction in relational data bases. In *Logic and Databases.* Plenum Press, 1978.

[Che76] P.P. Chen. The Entity-Relational Model- Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CKW92] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 1992.

[CMCG94] J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and DeMichiel L. G. Extending Relational Database Technology for New Applications. *IBM Systems Journal*, 33(2):264–279, 1994.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Communications of ACM*, 13(6):377–387, June 1970.

[Cod72] E.F. Codd. Relational Completeness of the Data Base Sublanguages. In *Data Base Systems, Courant Computer Science Symposia Series 6.* Prentice Hall, 1972.

[Cod79] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM TODS*, 4(4), December 1979.

[Cod81] E.F. Codd. Data Models in Database Management. *ACM SIGMOD Record*, 11(2), February 1981.

[Dat90] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1990.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In *2nd Int'l Workshop on Database Programming Languages*, pages 80–102, June 1989.

[Deu91] O. et al. Deux. The O2 System. *Communications of the ACM*, 34(10):35–48, October 1991.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies. In *Proceedings of the ACM SIGMOD Conference*, pages 356–367, May 1986.

[EW81] R. Elmasri and G. Wiederhold. GORDAS: A formal high-level query language for the entity-relationship model. In *Entity-Relationship Approach to Information Modeling and Analysis.* Ed. Elsevier, 1981.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM TOIS*, 5(1):48–69, 1987.

[Fis89] D.H. et al. Fishman. Overview of the Iris DBMS. In *Object-Oriented Concepts, Databases, and Applications*, chapter 10, pages 219–250. Addison-Wesley, 1989.

[FT83] P.C. Fisher and S.J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proc. of the 7th International Computer Software Applications Conference*, 1983.

[GLPS91] M. Guy, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to STARBURST: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, 1991.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[HFWC91] A. Heuer, J. Fuchs, U. Wiebking, and T.U. Clausthal. OSCAR: An Object-Oriented Database System with a Nested Relational Kernel. In *Entity-Relationship Approach: The Core of Conceptual Modeling. Proceedings of the Ninth International Conference*, pages 103–18, 1991.

[HM81] M. Hammer and D. McLeod. Database description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HR87] R. Hull and King R. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):140–173, September 1987.

[HY84] R. Hull and C. Yap. The Format Model: A Theory of Database Organization. *Journal of the Association for Computing Machinery*, 31(3):518–537, July 1984.

[HZ87] M. Hornick and S.B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[KBC+89] W. Kim, N. Ballou, H.-T. Chou, J.F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In *Object-Oriented Concepts, Databases, and Applications*, chapter 11, pages 251–282. Addison-Wesley, 1989.

[KC89] S.N. Khoshaflan and G.P. Copeland. Object Identity. In *Readings in Object-Oriented Database Systems*, pages 37–46. Morgan-Kaufmann, 1989.

[Kim] W. Kim. On Object-Oriented Database Technology. UniSQL, Inc., white paper.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[Kim92] W. Kim. On unifying relational and object-oriented database systems. In *ECOOP '92. European Conference on Object-Oriented Programming. Proceedings*, pages 1–18, 1992.

[Kim94] W. Kim. Observations on the ODMG-93 proposal for an object-oriented database language. *SIGMOD Record*, 23(1):4–9, March 1994.

[Kin89] R. King. My Cat is Object-Oriented. In *Object-Oriented Concepts, Databases, and Applications*, chapter 2, pages 23–30. Addison-Wesley, 1989.

[KL89] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *Proceedings of the ACM-SIGMOD Intl. Conference on Management of Data*, pages 134–146, June 1989.

[KR90] W. M. Kisworo and P. Rajagopalan. Implementation of an Object-Oriented Front-End to a Relational Database System. In *IEEE TENCON '90: 1990 IEEE Region 10 Conference on Computer and Communication Systems*, volume 2, pages 811–15, September 1990.

[Kul93] K.G. Kulkarni. Object-orientation and the SQL standard. *Computer Standards & Interfaces*, 15(2-3):287–300, July 1993.

[Las92] A.V. Lashmanov. Trends in development of database languages (relational and object-oriented languages and systems. *Automatic Documentation and Mathematical Linguistics*, 26(4):41–58, 1992.

[LH90] B. Lindsay and L. Haas. Extensibility in the Starburst Experimental Database System. In *Database Systems of the 90s. Int'l Symposium Proceedings*, pages 217–248, 1990.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):51–63, October 1991.

[LLPS91] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to STARBURST: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):95–109, October 1991.

[LRV89] C. Lecluse, P. Richard, and F. Velez. O2, an Object-Oriented Data Model. In *Readings in Object-Oriented Database Systems*, pages 227–236. Morgan-Kaufmann, 1989.

[Mai83] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relations in the relational data model. In *Proceedings of the 3rd International Conference on Very Large Databases*, pages 447–453, October 1977.

[MB90] T. Mason and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1990.

[Mos90] J.E.B. Moss. Design of the Mneme Persistent Object Store. *Transactions on Office Information Systems*, 8(2), April 1990.

[MR86] Stonebraker M. and L. Rowe. The Design of Postgres. In *Proceedings of the ACM SIGMOD Conference*, pages 340–355, May 1986.

[MR93] I.S. Mumick and K.A. Ross. Noodle: A Language for Declarative Querying in an Object-Oriented Database. In *Third Intl. Conference, DOOD '93. Proceedings*, pages 360–78, 1993.

[NCL+86] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, and M. Stanley. Implementation of a compiler for a semantic data model: experience with TAXIS. *SIGMOD Record*, 16(3):118–131, 1986.

[PBRV90] W. Premerlani, M. Blaha, J. Rumbaugh, and T. Varwig. An Object-Oriented Relational Database. *Communications of the ACM*, 33(11):99–108, Nov 1990.

[PD89] P. Pistor and P. Dadam. The Advanced Information Management Prototype. In *Nested Relations and Complex Objects in Databases. LNCS 361*, pages 3–26. Springer-Verlag, 1989.

[RM87] L. Rowe and Stonebraker M. The Postgres Data Model. In *Proc. of the XIII International Conference on Very Large Databases*, pages 289–300, 1987.

[SAB+89] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A Database Machine Based On Nested Relations. In *Nested Relations and Complex Objects in Databases. LNCS 361*, pages 27–49. Springer-Verlag, 1989.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, 1973.

[Shi81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):79–92, October 1991.

[SRL+90] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, Sep 1990.

[SS86] H.-J. Schek and H.M. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[SS89] H.-J. Schek and M.H. Scholl. The Two Roles of Nested Relations in the DASDBS Project. In *Nested Relations and Complex Objects in Databases. LNCS 361*, pages 50–68. Springer-Verlag, 1989.

[SS90] M. H. Scholl and H.-J. Schek. A Relational Object Model. In *INRIA. ICDT '90. Third International Conference on Database Theory. Proceedings*, pages 89–105, 1990.

[SS91] H.-J. Schek and M. H. Scholl. From Relations and Nested Relations to Object Models. In *Database systems of the 90s. International Symposium Proceedings*, pages 202–225, 1991.

[Sto] M. Stonebraker. Object-Relational Database Systems. Montage Software, Inc., white paper.

[Sto87] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 289–300, September 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Syb91a] Sybase, Inc., Emeryville, CA. *Open Client DB-Library/C Reference Manual*, 1991.

[Syb91b] Sybase, Inc., Emeryville, CA. *Transact-SQL User's Guide*, 1991.

[TZ84] S. Tsur and C. Zaniolo. An implementation of GEM-Supporting a semantic data model on a relational back-end. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 286–295, 1984.

[UhCLS94] S.D. Urban, Chiung hsun Chen Lai, and S. Saxena. The Design and Translation of ORL: An Object Retrieval Language. *Journal of Systems and Software*, 24(2):187–206, February 1994.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[US90] R. Unland and G. Schlageter. Object-Oriented Database Systems: Concepts and Perspectives. In *Database systems of the 90s. International Symposium Proceedings*, pages 154–97, 1990.

[WBC90] G. Wiederhold, T. Barasalou, and S. Chaudhuri. Managing Objects in a Relational Framework. Technical report, Computer Science Department, Stanford University, 1990.

[Wie91] R.J. Wieringa. A formalization of objects using equational dynamic logic. In *Second International Conference on Deductive and Object-Oriented Databases. Proceedings. LNCS 566*, pages 431–452, December 1991.

[WLH90] K. Wilkinson, P. Lyngbael, and W. Hasan. The Iris architecture and implementation. *ACM Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[ZM89a] S. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In *Readings in Object-Oriented Database Systems*, pages 1–32. Morgan-Kaufmann, 1989.

[ZM89b] S.B. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann, 1989.

Appendix A

Implemented syntax of COOL

The BNF Grammar for the COOL parser is:

```
<cool_list>             ::=
                        <cool> ;
                      | <cool_list> <cool> ;

<cool>                  ::=
                        <schema>
                      | <manipulative_statement>

<schema>                ::=
                        /* empty */
                      | <schema_element_list>

<schema_element_list>   ::=
                        <schema_element>
                      | <schema_element_list> <schema_element>

<schema_element>        ::=
                        <objectcls_def>

<objectcls_def>         ::=
                        CREATE OBJ[ECT] CL[AS]S <objectcls>
                                  ( <objectcls_element_commalist> )

<objectcls_element_commalist> ::=
                              <objectcls_element>
                            | <objectcls_element_commalist> ,
                                         <objectcls_element>

<objectcls_element>     ::=
                          <attribute_def>
                        | <objectcls_key_def>

<attribute_def>           ::=
                          <attribute> <attribute_type>

<objectcls_key_def>       ::=
                          CANDIDATE KEY ( <attribute_commalist> )
                        | PRIMARY KEY <attribute>
                        | SUPER KEY ( <relationship_field_commalist> )

<relationship_field_commalist> ::=
                                 <relationship_field>
                               | <relationship_field_commalist> ,
                                       <relationship_field>

<relationship_field> ::=
                     <attribute> ( <parent_objectcls> )
                   | <attribute>
                     ( <parent_objectcls> : <reference_attribute> )

<attribute_commalist> ::=
                      <attribute>
                    | <attribute_commalist> , <attribute>

/* data manipulation statements */

<manipulative_statement> ::=
                          <select_statement>
                        | <delete_statement>
                        | <insert_instance_statement>
                        | <insert_manyobjects_statement>
                        | <update_statement>
                        | <drop_class_statement>
                        | <dump_database_statement>

<insert_instance_statement> ::=
                     INSERT OBJ[ECT] INS[TANCE] INTO <objectcls>
                                  ( <insert_value_commalist> )
```

```
<insert_value_commalist> ::=
                          <insert_value>
                        | <insert_value_commalist> , <insert_value>

<insert_value>           ::=
                         <attribute> : <alpha>
                       | <attribute> : <numeric>

<insert_manyobjects_statement> ::=
                          INSERT INTO <objectcls> <select_statement>

<delete_statement>       ::=
                         <delete_uncond_statement>
                       | <delete_withcond_statement>

<delete_uncond_statement> ::=
                           DELETE ALL <from_clause>

<delete_withcond_statement> ::=
                             DELETE <from_clause> WHERE <condition>

<update_statement> ::=
                   UPDATE <objectcls> WHERE <condition>
                                      SET <update_value_commalist>

<update_value_commalist> ::=
                          <update_value>
                        | <update_value_commalist> , <update_value>

<update_value>           ::=
                         <attribute> : <scalar_expr>
                       | <attribute> : <alpha>

<drop_class_statement>   ::=
                         DROP <objectcls>

<dump_database_statement>        ::=
                                 DUMP DATABASE

<select_statement>       ::=
                        SELECT <selection> FROM <object_class>
                                    [WHERE <where_expression>]

<selection>             ::=
                          <attribute_commalist>
                        | *


<where_expression>      ::=
                          <condition> [<logical_xref_list>]
                        | <quantified_xreference> [<logical_xref_list>]

<logical_xref_list>     ::=
                          <logical_xref>
                        | <logical_xref_list> <logical_xref>

<logical_xref>          ::=
                          OR <quantified_xreference>
                        | AND <quantified_xreference>

<condition>             ::=
                          <relational_expr>
                        | <condition> AND <condition>
                        | <condition> OR <condition>
                        | ( <condition> )

<relational_expr>       ::=
                          <left_expr> COMPARISON <right_expr>

<left_expr>             ::=
                          <attribute>

<right_expr>            ::=
                          <scalar_expr>
                        | <alpha>

<scalar_expr>           ::=
                          <scalar_expr> + <scalar_expr>
                        | <scalar_expr> - <scalar_expr>
                        | <scalar_expr> * <scalar_expr>
                        | <scalar_expr> / <scalar_expr>
                        | ( <scalar_expr> )
                        | <numeric>

<quantified_xreference> ::=
                          <genitive_relation> ( <where_expression> )


<genitive_relation> ::=

/* Parent Child genitive relation where (1) is the formal syntax */
               <quantifier_pc> <parent> . <reference_attribute> *
                                                      <formal_child>
/* and (2) is the natural language syntax */
            |  <quantifier_pc> <parent>'S
                      [* <reference_attribute>] <natural_child>

/* Composite Parent Child genitive relation where
   (1) is the formal syntax */
            |  <quantifier_pc> <grandparent> .
                           <reference_attribute> * <parent> .
                                    <reference_attribute> '*' <child>
/* and (2) is the natural language syntax */
            |  <quantifier_pc> <grandparent>'S
                           [* <reference_attribute>] <parent>'S
                                    [* <reference_attribute>] <child>

/* Child Parent genitive relation where (1) is the formal syntax */
            |  <quantifier_cp> <child> . <child_superkey> * <parent>
/* and (2) is the natural language syntax */
            |  <quantifier_cp> <child>'S [<child_superkey>] <parent>

/* Composite Child Parent genitive relation where
   (1) is the formal syntax */
            |  <quantifier_cp> <child> . <child_superkey> *
                    <parent> '.' <parent_superkey '*' <grandparent>
/* and (2) is the natural language syntax */
            |  <quantifier_cp> <child>'S
                       [<child_superkey>] <parent>'S
                               [<parent_superkey>] <grandparent>
```

```
<child>                  ::=
                          <object_class>

<parent>                 ::=
                          <object_class>

<grandparent>            ::=
                          <object_class>

<formal_child>           ::=
                          <object_class>
                        | ( <object_class> ( <condition> ) )

<natural_child>          ::=
                          <object_class>
                        | ( <condition> ) <object_class>

<quantifier_pc>          ::=
                          FOR ALL
                        | FOR MOST
                        | FOR NONE
                        | FOR ALL BUT INTNUM
                        | FOR AT MOST INTNUM
                        | FOR AT LEAST INTNUM
                        | FOR MORE THAN INTNUM
                        | FOR LESS THAN INTNUM
                        | FOR EXACTLY INTNUM
                        | FOR NOT INTNUM

<quantifier_cp>          ::=
                          FOR THE
                        | FOR ITS
                        | FOR HER
                        | FOR HIS

<reference_attribute>    ::=
                          NAME

<alpha>                  ::=
                        STRING

<numeric>               ::=
                        INTNUM
                      | APPROXNUM

/* data types */

<attribute_type>        ::=
                        CHARACTER
                      | CHARACTER ( INTNUM )
                      | INTEGER

<object_class>          ::=
                        NAME

<attribute>             ::=
                        NAME

<child_superkey>        ::=
                        NAME

/*  COMPARISON:  = <> < > <= >=  */
```

The BNF Grammar for ERA parser is:

```
<era_list>              ::=
                        NL <era_expr>
                      | <era_list> NL <era_expr>

<era_expr>              ::=
                        /* empty */
                      | <join_statement>
                      | <select_statement>
                      | <group_select_statement>
                      | <intersect_statement>
                      | <union_statement>
                      | <subgroup_select_statement>
```

```
                       | <projection_statement>
                       | <pjoin_statement>

<join_statement> ::=
                <relation_name> = <relation_name>
                        ( <attribute> ) JOIN <relation_name>
                                                        ( <attribute> )

<pjoin_statement> ::=
                <relation_name> = <relation_name>
                       ( <attribute> ) PJOIN ( <variable_name> )
                                     <relation_name> ( <attribute> )

<select_statement> ::=
           <relation_name> = SELECT
                                ( <relation_name> ( <condition> ) )

<projection_statement> ::=
           <relation_name> = PROJECT
                                ( <relation_name> ( <selection> ) )

<group_select_statement> ::=
           <relation_name> = GROUP_SELECT ( <relation_name>
                                ( <quantifier_pc> <foreign_key>
                                           ( <condition_expr> ) ) )

<subgroup_select_statement> ::=
           <relation_name> = SUBGROUP_SELECT
                      ( <relation_name> ( <quantifier_pc>
                              ( <foreign_key> ( <condition> ) )
                                           ( <condition_expr> ) ) )

<condition_expr>         ::=
                         <condition>
                       | <condition> AND <variable_name>
                       | <condition> OR <variable_name>
                       | <variable_name>

<union_statement>        ::=
           <relation_name> = <relation_name> UNION <relation_name>
```

```
<intersect_statement>   ::=
           <relation_name> = <relation_name> INTERSECT
                                 ( <join_field> ) <relation_name>

<selection>             ::=
                        <attribute_commalist>
                      | *

<attribute_commalist>   ::=
                        <attribute>
                      | <attribute_commalist> , <attribute>

<condition>             ::=
                        <relational_expr>
                      | <condition> AND <condition>
                      | <condition> OR <condition>
                      | ( <condition> )

<relational_expr>       ::=
                        <left_expr> COMPARISON <right_expr>

<left_expr>             ::=
                        <attribute>

<right_expr>            ::=
                        <scalar_expr>
                      | <alpha>

<scalar_expr>           ::=
                        <scalar_expr> + <scalar_expr>
                      | <scalar_expr> - <scalar_expr>
                      | <scalar_expr> * <scalar_expr>
                      | <scalar_expr> / <scalar_expr>
                      | ( <scalar_expr> )
                      | <numeric>

<quantifier_pc>         ::=
                        FOR ALL
                      | FOR MOST
                      | FOR NONE
                      | FOR ALL BUT INTNUM
                      | FOR AT MOST INTNUM
                      | FOR AT LEAST INTNUM
                      | FOR MORE THAN INTNUM
                      | FOR LESS THAN INTNUM
                      | FOR EXACTLY INTNUM
                      | FOR NOT INTNUM

<alpha>                 ::=
                        STRING

<numeric>               ::=
                        INTNUM
                      | APPROXNUM

<relation_name>         ::=
                        NAME

<variable_name>         ::=
                        VAR

<foreign_key>           ::=
                        NAME

<join_field>            ::=
                        NAME

<attribute>             ::=
                        NAME
```

Appendix B

The aircraft maintenance database schema

```
Class Airline

  Properties:

    AL#         string,
    hqadd       string,
    emp_num     int,
    Airclist    set of Aircraft,
    Ownelist_A  set of Ownerships.
```

```
Class Aircraft

  Properties:

   AC#         string,
   type        string,
   fabricant   string,
   AL#         Airline,
   Partlist_A  set of Parts_on_Board,
   Servlist    set of Sevices.
```

```
Class Depot_Ownership

  Properties:

   O#    string,
   share   int,
   MD#   Maintenance_Depot,
   AL#   Airline.
```

```
Class Maintenance_Depot

  Properties:

   MD#         string,
   address     string,
   Ownelist_M  set of Services,
   Servlist_M  set of Services,
   Techlist    set of Technicians,
   Partlist_M  set of PartType_Inventory.
```

```
Class Service

  Properties:
```

```
Class Service_Project

  Properties:
    SV#         string,
    description string,
    Servlist_S  set of Service_Project,
    AC# Aircraft,
    MD# Maintenance_Depot.
```

```
    SVP#        string,
    description string,
    Joblist_S   set of Jobs,
    SV#         Service.
```

```
Class Parts_on_Board
  Properties:
    PB#         string,
    part#       string,
    status      string,
    AC#         Aircraft,
    PT#         PartType_Inventory,
    S#          Supplier.
```

```
Class Job
  Properties:
    J#          string,
    descr       string,
    start       date,
    finish      date,
    status      string,
    SVP#        Service_Project,
    T#          Technician.
```

```
Class Technician
  Properties:
    T#          string,
    name        string,
    title       string,
    Joblist_T set of Jobs,
    MD#         Maintenance_Depot
```

```
Class PartType_Inventory
  Properties:
    PT#             string,
    typename        string,
    qty             int,
    Partlist_P set of Parts_on_Board,
    Shiplist_P set of Shipment_Data,
    Inner_Parts   set of Constructs,
Outer_Parts   set of Constructs,
MD#           Maintenance_Depot.
```

```
Class Supplier
  Properties:
    S#           string,
    address      string,
    Partlist_S   set of Parts_on_Board,
    Shiplist_S   set of Shipment_Data.
```

```
Class Shipment_Data
  Properties:
    SH#        string,
    price      int,
    qty        int,
    S#         Supplier,
    PT#        PartType_Inventory.
```

```
Class Construct
  Properties:
    C#         string,
    location   string,
    PT_outer   PartType_Inventory,
    PT_inner   PartType_Inventory.
```

Appendix C

Aircraft maintenance database definition

C.1 COOL definition of the example database

```
create object class Airline
( AL# CHAR(2),
hqadd    char(20),
emp_num  int,
primary key AL#);

create obj cl Aircraft
(AC#  char(4),
AL# char(2),
fabricant char(20),
type char(4),
primary key AC#,
super key (AL# (Airline)));

create obj cl Depot_Ownership
(O# char(4),
AL# char(2),
MD# char(4),
share   int,
primary key O#,
super key (MD# (Maintenance_Depot), AL# (Airline)));

create obj cl Service
(SV#    char(4),
AC#     char(4),
MD# char(4),
description char(20),
primary key SV#,
super key (AC# (Aircraft), MD# (Maintenance_Depot)));
```

```
create object class Service_Project
( SVP#   char(4),
SV#      char(4),
description char(20),
primary key SVP#,
super key (SV# (Service)));

create obj cl Maintenance_Depot
(MD#    char(4),
address     char(20),
primary key MD#);

create obj cl Parts_on_Board
(PB# char(4),
AC#    char(4),
PT#     char(4),
S# char(4),
part# char(4),
status char(2),
primary key PB#,
super key (AC# (Aircraft), PT# (PartType_Inventory),
           S# (Supplier)));

create obj cl Job
(J#     char(4),
T#    char(4),
SVP#     char(4),
descr   char(10),
start char (10),
finish char(10),
status char(3),
PT# char(4),
part# char(4),
primary key J#,
super key (SVP# (Service_Project), T# (Technician)));

create obj cl Technician
(T#     char(4),
MD#    char(4),
name    char(20),
title char (15),
primary key T#,
super key (MD# (Maintenance_Depot)));

create obj cl PartType_Inventory
(PT# char(4),
MD#    char(4),
typename char(20),
qty  int,
primary key PT#,
super key (MD# (Maintenance_Depot)));


create obj cl Supplier
(S# char(4),
address char(20),
primary key S#);

create obj cl Shipment_Data
(SH# char(4),
PT# char(4),
S# char(4),
price int,
qty int,
primary key SH#,
super key (PT# (PartType_Inventory), S# (Supplier)));

create obj cl Construct
(C# char(4),
PT_outer char(4),
PT_inner char(4),
location char(10),
primary key C#,
super key (PT_outer (PartType_Inventory:inner_parts),
PT_inner (PartType_Inventory:outer_parts)));
```

C.2 SQL definition of the example database

```
create table Airline (
OID int,
AL# char (2),
hqadd char (20),
emp_num int,
)
create table Aircraft (
OID int,
AC# char (4),
AL# char (2),
fabricant char (20),
type char (4),
)
create table Depot_Ownership (
OID int,
O# char (4),
AL# char (2),
MD# char (4),
share int,
)
create table Service (
OID int,
SV# char (4),
AC# char (4),
MD# char (4),
description char (20),
)
create table Service_Project (
OID int,
SVP# char (4),
SV# char (4),
description char (20),
)
create table Maintenance_Depot (
OID int,
MD# char (4),
address char (20),
)
create table Parts_on_Board (
OID int,
PB# char (4),
AC# char (4),
PT# char (4),
S# char (4),
part# char (4),
status char (2),
)
create table Job (
OID int,
J# char (4),
T# char (4),
SVP# char (4),
descr char (10),
start char (10),
finish char (10),
status char (3),
PT# char (4),
part# char (4),
)
create table Technician (
OID int,
T# char (4),
MD# char (4),
name char (20),
title char (15),
)
create table PartType_Inventory (
OID int,
PT# char (4),
MD# char (4),
typename char (20),
qty int,
)
create table Supplier (
OID int,
S# char (4),
address char (20),
)
create table Shipment_Data (
OID int,
SH# char (4),
PT# char (4),
S# char (4),
price int,
qty int,
)
create table Construct (
OID int,
C# char (4),
PT_outer char (4),
PT_inner char (4),
location char (10),
)
```

Appendix D

Testing the Aircraft Maintenance Database

D.1 Create and Load the database

The Aircraft Maintenance Database can be created and loaded using the procedure loadAMD, as follows:

```
*** Script initiated by Carmen Rata on Fri Jan 20 20:13:37 1995
[cool] >>?
[cool] >>? cat loadAMD
#!/bin/sh

isql -Urata -Pcarmen < clean.sql
isql -Urata -Pcarmen < init.sql
coo create.cool
echo The database was created
coo insertAirline.cool
echo Airline class loaded
coo insertMD.cool
echo Maintenance_Depot class loaded
coo insertPTI.cool
echo PartType_Inventory class loaded
coo insertSup.cool
echo Supplier class loaded
coo insertAircraft.cool
echo Aircraft class loaded
coo insertT.cool
echo Technician class loaded
coo insertServ.cool
echo Service class loaded
coo insertServP.cool
echo Service_Project class loaded
```

```
coo insertJob.cool
echo Job class loaded
coo insertShipD.cool
echo Shipment_Data class loaded
coo insertPB.cool
echo Parts_on_Board class loaded

echo The database was created and loaded successfully!
```

```
[cool] >>? loadAMD
The database was created
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Airline class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Maintenance_Depot class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
PartType_Inventory class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Supplier class loaded
(1 row affected)
(1 row affected)
```

```
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Aircraft class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Technician class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Service class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Service_Project class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Job class loaded
(1 row affected)
```

```
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Shipment_Data class loaded
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Parts_on_Board class loaded
The database was created and loaded successfully!
[cool] >>? exit
exit

*** Script completed on Fri Jan 20 20:15:46 1995
*** Script session length is 135 lines
19434D540CBD35A1B11DEA4531C9BCEF24989D943C997E543EA2AB54F13D66BA8
```

D.2 Contents of test database

```
*** Script initiated by Carmen Rata on Fri Jan 20 20:10:03 1995
[cool] >>?
[cool] >>? cat dbcontents.sql
print "Airline"
print ""
select * from Airline
print ""
print "Maintenance_Depot"
print ""
select * from Maintenance_Depot
print ""
print "PartType_Inventory"
print ""
select * from PartType_Inventory
print ""
print "Supplier"
print ""
select * from Supplier
print ""
print "Aircraft"
print ""
select * from Aircraft
print ""
print "Technician"
print ""
select * from Technician
print ""
print "Service"
print ""
select * from Service
print ""
print "Service_Project"
print ""
select * from Service_Project
print ""
print "Job"
print ""
select * from Job
print ""
print "Shipment_Data"
print ""
select * from Shipment_Data
print ""
print "Parts_on_Board"
print ""
select * from Parts_on_Board
[cool] >>?
[cool] >>? isql
Password:
1> :r dbcontents.sql
45> go
Airline

 OID         AL# hqadd                emp_num
 ----------- --- -------------------- -----------
           1 RO  Bucharest                   3000
           2 AL  Rome                        6000
           3 AF  Paris                       7000
           4 BA  London                     10000

(4 rows affected)

Maintenance_Depot

 OID         MD#  address
 ----------- ---- --------------------
           1 M1   Montreal
           2 M2   Boston
           3 M3   Calgary
           4 M4   San Jose
           5 M5   San Diego

(5 rows affected)

PartType_Inventory

 OID         PT#  MD#  typename             qty
 ----------- ---- ---- -------------------- -----------
           1 P1   M1   dial                           2
           2 P4   M1   bolt                        2000
           3 P6   M2   valve                         40
           4 P9   M3   nut                          600
           5 P44  M2   turbine                        4
           6 P7   M1   pipe                        4000
```

```
          7 P11  M1   fastener                          1000
          8 P19  M4   lever                                3
          9 P10  M4   seat                               400

(9 rows affected)

Supplier

 OID         S#   address
 ----------- ---- --------------------
           1 S1   Seattle
           2 S2   Los Angeles
           3 S3   Vancouver
           4 S4   San Francisco

(4 rows affected)

Aircraft

 OID         AC#  AL# fabricant            type
 ----------- ---- --- -------------------- ----
           1 AB41 AF  Airbus               A320
           2 IR10 AF  McDonald_Douglas     DC10
           3 BC01 AL  Boeing               B737
           4 BC11 AL  Boeing               B727
           5 BC18 AL  Boeing               B727
           6 AB50 BA  Airbus               B320
           7 AB10 BA  Airbus               A320

(7 rows affected)

Technician

 OID         T#   MD#  name                 title
 ----------- ---- ---- -------------------- ---------------
           1 t1   M1   Smith                engineer
           2 t2   M2   Brown                technician
           3 t3   M3   Victor               analyst
           4 t4   M1   Green                mechanic
           5 t5   M1   Taylor               electrician
          6 t6   M3   Jones                engineer

(6 rows affected)

Service

 OID         SV#  AC#  MD#  description
 ----------- ---- ---- ---- --------------------
           1 SV2  BC18 M1   computer repair
           2 SV1  BC01 M1   autopilot checking
           3 SV3  BC11 M2   engine overhaul
           4 SV4  BC01 M3   metal fatigue

(4 rows affected)

Service_Project

 OID         SVP# SV#  description
 ----------- ---- ---- --------------------
           1 SVP1 SV1  check1
           2 SVP2 SV1  check2
           3 SVP3 SV1  check3
           4 SVP4 SV3  engine1
           5 SVP5 SV3  engine2
           6 SVP6 SV2  printer
           7 SVP7 SV2  disk drive
           8 SVP8 SV4  wing
           9 SVP9 SV4  tail

(9 rows affected)

Job

 OID         J#   T#   SVP# descr      start      finish     status PT#  part#
 ----------- ---- ---- ---- ---------- ---------- ---------- ------ ---- -----
           1 J2   t1   SVP1 task a     11/10/94   14/10/94   ok     P4   12
           2 J1   t2   SVP2 task b     10/1/95    30/1/95    ver    P4   167
           3 J3   t6   SVP2 task f     3/1/95     4/1/95     ok     P4   543
           4 J4   t5   SVP3 task c     1/9/94     14/10/94   ok     P6   122
           5 J5   t6   SVP3 task g     11/9/94    12/9/94    ok     P1   127
           6 J6   t1   SVP3 task e     12/9/94    14/9/94    ok     P6   521
           7 J7   t4   SVP4 task 12    1/2/95     1/3/95     de     P44  12
```

```
         8 J8    t4   SVP5 task 3     1/12/94    10/1/95    de     P44  45
         9 J9    t2   SVP6 task z     14/12/94   15/12/94   ok     P4   732
        10 J10   t4   SVP8 task s     1/8/94     14/19/94   ok     P9   768

(10 rows affected)

Shipment_Data

 OID         SH#  PT#  S#   price       qty
 ----------- ---- ---- ---- ----------- -----------
           1 SH1  P4   S2            45         300
           2 SH2  P4   S2           100        1000
           3 SH3  P11  S2            38         200
           4 SH7  P11  S2            38        2000
           5 SH4  P1   S1            20        1000
           6 SH5  P6   S1          6800           2
           7 SH6  P10  S3           700           3

(7 rows affected)

Parts_on_Board

 OID         PB#  AC#  PT#  S#   part# status
 ----------- ---- ---- ---- ---- ----- ------
           1 X21  BC01 P1   S1   123   ok
           2 X51  BC01 P4   S4   723   ok
           3 X44  BC01 P6   S1   163   ok
           4 X34  BC01 P9   S1   873   ok
           5 A21  BC11 P44  S1   23    ok
           6 A45  BC11 P7   S1   923   ok
           7 A71  BC11 P11  S3   33    de
           8 k61  BC18 P19  S4   122   ok
           9 k55  BC18 P1   S1   234   ok
          10 k89  BC18 P10  S2   56    ok
          11 G46  AB10 P11  S1   456   ok
          12 G47  AB10 P44  S2   156   de
          13 T49  AB50 P10  S3   556   ok
          14 T62  AB50 P7   S4   567   de
          15 T34  AB50 P19  S1   96    ok

(15 rows affected)
```

```
1> exit
[cool] >>? exit
exit

*** Script completed on Fri Jan 20 20:10:57 1995
*** Script session length is 214 lines
108F98CF795BDA8AD46B0338316EB24B83A9B065C44684811F61EF2E909D5FEC5A
```

D.3 Example of use of the COOL insert command

In the following example we insert a new instance in the object class Airline.

```
*** Script initiated by Carmen Rata on Fri Jan 20 20:26:21 1995
[cool] >>? isql
Password:
1> select * from Airline
2> go
 OID         AL# hqadd                emp_num
 ----------- --- -------------------- -----------
           1 RO  Bucharest                   3000
           2 AL  Rome                        6000
           3 AF  Paris                       7000
           4 BA  London                     10000

(4 rows affected)
1> exit
[cool] >>? cat insert_ex.cool
insert obj ins into Airline
              ( AL# : "AA", hqadd : "Dallas", emp_num : 10000);
[cool] >>?
[cool] >>? coo insert_ex.cool
(1 row affected)
[cool] >>?
[cool] >>? isql
Password:
1> select * from Airline
2> go
```

```
 OID         AL# hqadd                emp_num
 ----------- --- -------------------- -----------
           1 RO  Bucharest                   3000
           2 AL  Rome                        6000
           3 AF  Paris                       7000
           4 BA  London                     10000
           5 AA  Dallas                     10000

(5 rows affected)
1> exit
[cool] >>? exit
exit

*** Script completed on Fri Jan 20 20:28:14 1995
*** Script session length is 40 lines
3E4CE9100669680757AC4E9FBF297A847D7FAE3A0BCC723D6FADDF44CD45F3FF5
```

D.4 Examples of COOL query executions

```
*** Script initiated by Carmen Rata on Fri Jan 20 20:34:40 1995
[cool] >>?
[cool] >>? cat t1.cool
-- List airline code and headquarters location for airlines where
-- most aircraft of type Boeing have all parts on board with
-- status OK.

select AL#, hqadd from Airline
where for most Airline's (fabricant = "Boeing") Aircraft
   (for all Aircraft's Parts_on_Board (status = "ok"));
[cool] >>?
[cool] >>? coo t1.cool
(4 rows affected)
(4 rows affected)
(7 rows affected)
(3 rows affected)
(1 row affected)
(1 row affected)
```

```
(1 row affected)
 AL# hqadd
 --- --------------------
 AL  Rome

(1 row affected)
[cool] >>?
[cool] >>? cat t2.cool

-- Give the PT# and quantity for each type of part in inventory that
-- has (a) at least 2 shipments from supplier in Los Angeles , and
-- (b) has status 'defect' on at least one aircraft on which it is
-- used.

select PT#, qty from PartType_Inventory
where for at least 2 PartType_Inventory's Shipment_Data
   (for its Shipment_Data's Supplier (address = "Los Angeles"))
and for at least 1 PartType_Inventory's Parts_on_Board
                                                (status = "de");
[cool] >>?
[cool] >>? coo t2.cool
(1 row affected)
(1 row affected)
(4 rows affected)
(4 rows affected)
(4 rows affected)
(7 rows affected)
(2 rows affected)
(3 rows affected)
(1 row affected)
(1 row affected)
(1 row affected)
 PT#  qty
 ---- -----------
 P11         1000

(1 row affected)
[cool] >>?
[cool] >>? cat t3.cool
```

```
-- What maintenance depot in Montreal has carried out at least two
-- sevice projects in each of which all jobs involved part type 'P4'.

select MD# from Maintenance_Depot
where address = "Montreal"
and for at least 2 Maintenance_Depot's Service's Service_Project
        (for all Service_Project's Job (PT# = "P4"));
[cool] >>?
[cool] >>? coo t3.cool
(5 rows affected)
(5 rows affected)
(9 rows affected)
(4 rows affected)
(9 rows affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
 MD#
 ----
 M1

(1 row affected)
[cool] >>? exit
exit

*** Script completed on Fri Jan 20 20:37:11 1995
*** Script session length is 88 lines
144BFEB9F3F91924E222FC39FD7CCF27D37A8D48FBA35DCEB0F9842A656309CE5
```

The equivalent SQL queries for the three COOL test queries will give the following results when executed against the same database.

```
*** Script initiated by Carmen Rata on Tue Jan 24 00:06:24 1995
[cool] >>?
[cool] >>? cat tt1.sql
select AL#, hqadd from Airline
where (select count (*)
       from Aircraft
       where fabricant = "Boeing"
       and Aircraft.AL# = Airline.AL#
       and AC# not in (select AC# from Parts_on_Board
                       where status <> "ok")
       )
       >
      (select count (*)
       from Aircraft
       where fabricant = "Boeing"
       and Aircraft.AL# = Airline.AL#
       and AC# in (select AC# from Parts_on_Board
                       where status <> "ok")
       )
[cool] >>?
[cool] >>? isql
Password:
1> :r tt1.sql
36> go
 AL# hqadd
 --- --------------------
 AL  Rome

(1 row affected)
1>
2> exit
[cool] >>? cat tt2.sql
select PT#, qty from PartType_Inventory
where (select count (*)
       from Shipment_Data
       where PartType_Inventory.PT# = Shipment_Data.PT#
       and S# in (select S# from Supplier
                  where address = "Los Angeles") ) >=2
and (select count (*)
     from Parts_on_Board
     where PartType_Inventory.PT# = Parts_on_Board.PT#
     and status = "de")>=1
[cool] >>?
[cool] >>? isql
Password:
1> :r tt2.sql
12> go
 PT#  qty
 ---- -----------
 P11         1000

(1 row affected)
1> exit
[cool] >>?
[cool] >>? cat tt3.sql
select MD# from Maintenance_Depot
where address ="Montreal"
and (select count (*)
     from Service, Service_Project
     where Service.MD# = Maintenance_Depot.MD#
     and Service.SV# = Service_Project.SV#
     and SVP# not in (select SVP# from Job
                      where PT# <> "P4")) >=2


[cool] >>? idΦ[Ksql
Password:
1> :r tt3.sql
12> go
 MD#
 ----
 M1

(1 row affected)
1> exit
[cool] >>? exit
exit

*** Script completed on Tue Jan 24 00:08:28 1995
*** Script session length is 99 lines
```

```
C0B1D4D74D7A127C27B191CACCB013F4B6FAD27617354DC68ED01661AE4EB6F6B
```

D.5 Example using the update statement

In the following example, we update an attribute of an Airline instance.

```
*** Script initiated by Carmen Rata on Thu Jan 19 14:18:04 1995
[cool] >>?
[cool] >>? cat update.cool
update Airline where AL# = "RO" set hqadd : "Arad";
[cool] >>?
[cool] >>? coo update.cool
 OID         AL# hqadd                          emp_num
 ----------- --- ------------------------------ -----------
           1 RO  Bucharest                             3000

(1 row affected)
(1 row affected)
[cool] >>?
[cool] >>? isql
Password:
1> select * from Airline
2> go
 OID         AL# hqadd                          emp_num
 ----------- --- ------------------------------ -----------
           2 AL  Rome                                  6000
           3 AF  Paris                                 7000
           4 BA  London                               10000
           5 AA  Dallas                               10000
           1 RO  Arad                                  3000

(5 rows affected)
1> exit
[cool] >>? exit
exit
```

```
*** Script completed on Thu Jan 19 14:19:27 1995
*** Script session length is 33 lines
36D4B3604D1922173DA36DCC855CC4AD2C055C2E525EBB8D70F887D23FBA8BF3B06
```

D.6 Example using the delete statement

The "delete with condition" statement will be executed only for an instance without any descendents. In the following example we try to delete an instance of the class Shipment_Data.

```
*** Script initiated by Carmen Rata on Thu Jan 19 15:37:34 1995
[cool] >>?
[cool] >>? isql
Password:
1> select * from Shipment_Data
2> go
 OID         SH#  PT#  S#   price       qty
 ----------- ---- ---- ---- ----------- -----------
           1 SH1  P4   S2            45         300
           2 SH2  P4   S2           100        1000
           3 SH3  P11  S2            38         200
           4 SH7  P11  S2            38        2000
           5 SH4  P1   S1            20        1000
           6 SH5  P6   S1          6800           2
           9 SH6  P10  S3           700           3

(7 rows affected)
1> exit
[cool] >>?
[cool] >>? cat deleteShip.cool
delete from Shipment_Data where SH# = "SH6";
[cool] >>?
[cool] >>? coo deleteShip.cool
(1 row affected)
[cool] >>?
```

```
[cool] >>? isql
Password:
1> select * from Shipment_Data
2> go
 OID         SH#  PT#  S#   price       qty
 ----------- ---- ---- ---- ----------- -----------
           1 SH1  P4   S2            45         300
           2 SH2  P4   S2           100        1000
           3 SH3  P11  S2            38         200
           4 SH7  P11  S2            38        2000
           5 SH4  P1   S1            20        1000
           6 SH5  P6   S1          6800           2

(6 rows affected)
1> exit
[cool] >>? exit
exit

*** Script completed on Thu Jan 19 15:38:53 1995
*** Script session length is 46 lines
DF8F75826F637AA5DA0B11AF39E1FC52D3D1D7CC044890E5648FCC2224D5B271FBD3
```

D.7 Typical error messages

* An error message will be generated if a child instance is going to be inserted but the parent instance is not found in the database.

```
*** Script initiated by Carmen Rata on Thu Jan 19 19:53:42 1995
[cool] >>?
[cool] >>? isql
Password:
1> select * from Maintenance_Depot
2> go
 OID         MD#  address
 ----------- ---- ------------------------------
           1 M1   Montreal
           2 M2   Boston
           3 M3   Calgary
           4 M4   San Jose
           5 M1   San Diego

(5 rows affected)
1> exit
[cool] >>?
[cool] >>? cat insertTech.cool
insert obj ins into Technician
     ( T# : "t10", MD# : "M10", name : "Oreste", title : "musician");
[cool] >>?
[cool] >>? coo insertTech.cool

Class Maintenance_Depot has no instance with primary key =  "M10"
You try to insert a child without a parent  !!!
[cool] >>?
[cool] >>? isql
Password:
1> select * from Maintenance_Depot
2> go
 OID         MD#  address
 ----------- ---- ------------------------------
           1 M1   Montreal
           2 M2   Boston
           3 M3   Calgary
           4 M4   San Jose
           5 M1   San Diego

(5 rows affected)
1>
2> exit
[cool] >>? exit
exit

*** Script completed on Thu Jan 19 19:55:27 1995
*** Script session length is 46 lines
214F1A588753DA79C95F7322C1501B0D553EED47436C776423BA2D780D125F313
```

* An error message will be generated if instances with descendents are attempted to be deleted.

```
*** Script initiated by Carmen Rata on Thu Jan 19 20:22:46 1995
[cool] >>? isql
Password:
1> select * from Airline
2> go
 OID         AL# hqadd                          emp_num
 ----------- --- ------------------------------ -----------
           2 AL  Rome                                  6000
           3 AF  Paris                                 7000
           4 BA  London                               10000
           5 AA  Dallas                               10000
           1 RO  Arad                                  3000

(5 rows affected)
1> exit
[cool] >>?
[cool] >>? cat delete.cool
delete from Airline where AL# = "BA";
[cool] >>?
[cool] >>? coo delete.cool

DELETE from Airline CAN NOT be performed!!!

There are descendents out there !!!
[cool] >>?
[cool] >>? isql
Password:
1> select * from Airline
2> go
 OID         AL# hqadd                          emp_num
 ----------- --- ------------------------------ -----------
           2 AL  Rome                                  6000
           3 AF  Paris                                 7000
           4 BA  London                               10000
           5 AA  Dallas                               10000
           1 RO  Arad                                      3000

(5 rows affected)
1> exit
[cool] >>? exit
exit

*** Script completed on Thu Jan 19 20:23:59 1995
*** Script session length is 45 lines
2F0E74EF57DE9260F3D6877929C936FECCCE0F45DBAB1E7C91F5FD6A5281AA8BF3
```

* An error message will be generated when a multiple level quantifier expression has not matched genitive relations.

```
*** Script initiated by Carmen Rata on Thu Jan 19 20:36:01 1995
[cool] >>? cat q23.cool
-- Get full details of an Airline with headquaters located in
-- San Diego where most of its aircraft have (a) Boeing as a
-- manufacturer and (b) at least 1 scheduled 'computer repair'
-- service_project.

select * from Airline
where HQlocation = "San Diego"
and for most Airline's Aircraft ( fabricant = "Boeing"
                    and for at least 1 Service's Service_Project
                             ( description = "computer repair"));
[cool] >>?
[cool] >>? coo q23.cool
 Unmatched nested quantified expression
(0 rows affected)
 SV#
 ----

(0 rows affected)
[cool] >>? exit
exit
```

```
*** Script completed on Thu Jan 19 20:37:09 1995
*** Script session length is 23 lines
226E26DA12FC94FB4611AE1AA4F9E7D6EF68BB76C89919AE9A6C49DC54913D61F
```

D.8 Command 'coo'

The contents of the script file 'coo' that performs the translation and the execution of COOL expressions is as follows:

```
*** Script initiated by Carmen Rata on Sat Jan 21 23:31:10 1995
[cool] >>?
[cool] >>? cat coo
#!/bin/sh
name=`echo $1 | sed 's/\..*//'`
rm -f ${name}.era ${name}.sql
cool ${name}.cool
if [ -s ${name}.era ]; then
        era ${name}.era
fi

isql -Urata -Pcarmen < ${name}.sql
[cool] >>? exit
exit

*** Script completed on Sat Jan 21 23:31:27 1995
*** Script session length is 20 lines
D44D224F03596BB175896D9BBF17F8CAD1E890651BF74AB642F28AC1A89DD2A8ED
```

Appendix E

OOPL Database Systems

This approach has OO programming languages as a starting point.

E.1 Concepts

The concepts that describe the OOPL approach to OO data models are best formalized in the Object-Oriented Database System Manifesto [ABD+89] and in [Ban93]. These concepts are distinct from those of the other approaches:

- The notion of encapsulation becomes mandatory. The original definition of encapsulation provided by the OO paradigm is that procedures are public, whereas data is private. However, this concept is often too restrictive for OODBMSs [Cat91], and there are as many different variations of encapsulation between the OODBMSs as there are between programming languages. The different kinds of encapsulation vary by the degree in which either data or methods may be in the public and private portions of an object class.

- The concept of polymorphism is associated with overriding, overloading and late binding. The "is a" nature of inheritance is tightly coupled with the idea of polymorphism (the ability to take more than one form).In an OOPL, a polymorphic reference is one that can, over time, refer to instances of more than one class. In the OODBMS world there is a classic example of a polymorphic method: the function 'display', that receives an object as an input and performs the display of the object on the screen [BM93]. A user might want to apply uniformly the function display to a variety of objects: text, graphic, map, value. The operation has a single name and can be defined in a more general class. However, the implementation of display is redefined for each of the subclasses. The redefinition is called **overriding**. The use of a single name for different programs is called **overloading**. Also, the system cannot bind the names of the class to the corresponding method at compile-time, but must do so during run-time. This translation is called **late binding**.

- The computational completeness of the query language is mandatory. Relational query languages alone are not computationally complete and relational query language expressions need to be embedded in host programming languages in order to achieve computational completeness.

- An ad hoc query facility must be provided but is not necessarily in the form of a query language, for example a graphical browser could be sufficient.

The OO data model maps directly onto the data types used by OO programming languages, most commonly those of C++ [Str86] and Smalltalk [GR83].

At the time of writing, as at the time when the first manifesto was written [ABD+89], the OOPL approach to database systems still has neither a common data model, nor a formal foundation. However, a wide variety of systems have been built. However, their commercial use remains minor compared with that of relational systems.

E.2 OO Database Systems that implement the approach

The OODBMS that implement the OOPL approach are built as extensions of an object-oriented programming language. The implemented systems of "OO Database programming languages", as Cattell calls them in [Cat91], vary in their choice of the programming language base and the query language. We have the following main groups:

1. **OODBMS based on C++.** The main examples are: ONTOS (Ontologic), ObjectStore (Object Design) [LLOW91], Objectivity/DB (Objectivity), VERSANT (Versant).

2. The O_2 OODBMS. O_2 is a system based on a derivative of C++, called CO2. The O_2 OODBMS [BBB+88, Cat91, Deu91] was developed by the Altair research group in France, in late 1986. It was implemented in C on top of an enhanced version of the Wisconsin Storage Manager (WiSS), which serves as disc manager. The originality of O_2 data model [LRV89] is the distinctive use of the concepts type and class. Value is an instance of a type and object is an instance of a class. An object is a pair (identifier, value). A value can be either an atom or basic type, such as, integer, float, string or boolean, or a value can be a structure, such as tuple, set, or list. Constructed types are inferred from basic types by means of recursive application of constructors tuple and set. Types are organized in an inheritance (or subtype) hierarchy enabling objects to share common structures and methods. A class is defined by its name, by the type of its instances and by the methods applicable to them. O_2 has been integrated with many programming languages, including C++, Lisp, and

Basic.

3. **OODBMS based on a OO version of Common Lisp.** The main example is ORION built at MCC in Austin, Texas. It has a commercial version ITASCA [BCG+87, KBC+89, Cat91]. ORION is intended to support OO applications in CAD/CAM, AI and OIS domains.

   ORION has been implemented in COMMOM LISP in order to be closely coupled to AI/Knowledge Base System applications, usually implemented in COMMON LISP. The application interface to ORION is an OO extension to LISP.

4. **OODBMS based on Smalltalk.** The main example is GemStone from Servio Logic [BMO+89, BOS91]. It uses Smalltalk as a data model. OPAL is the OO database language used for data definition and data manipulation and it is, as expected, an extension of Smalltalk. It allows path expressions, such as: anEmp.name.first. The query language is more like a limited calculus sublanguage in which queries are viewed as procedural OPAL code. GemStone is also integrated with C++.

Other systems that provide object data management but do not fit the major classification described in this thesis are:

- **Database system generators.** They are ODMS tailored to particular needs, typically with a custom data model and database language. The best known database system generators are EXODUS [CDG+89, Cat91], which provides a versatile storage manager for developing application specific database systems, and GENESIS [ZM89b, Cat91], which focuses more on automatic generation of DBMS modules from high-level architecture descriptions supplied by the database implementor.

- **Object Managers.** They are systems that basically provide a minimal persistent object store with concurrency control, and generally without a query and programming language. Some examples of object-manager approaches are the systems: Mneme, from the University of Massachusetts [Mos90], ObServer, at Brown University [HZ87], and WiSS, from the University of Wisconsin [CDKK85].

- **Object Front-End on top of an existing RDBMS.** PENGUIN [WBC90], which is an OO layer on top of a RDBMS through sophisticated multirelation views, is an example. In [PBRV90] there is a description of a technique for constructing an object-oriented DBMS (OO-DBMS) from a RDBMS, an OOPL and an object-oriented modeling technique; the programmer sees an object-oriented language with certain predefined operations that allows objects to be retrieved from and stored in a relational database.