Query Processing Issues in Object-Oriented Knowledge Base Systems

M. Tamer Özsu
Dave D. Straube*
Randal Peters

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

1 Introduction

The integration of database management (DB) and artificial intelligence (AI) technologies leading to the development of *knowledge base management systems* (KBMS) has been frequently discussed. The potential benefits of such an integration are significant. However, examples of successful AI/DB integration are not very common. The difficulty of finding a suitable integration architecture is one of the problems. Another problem is the unsuitability of the current database models and techniques for this integration.

The current commercial state-of-the-art in database technology is settling on the relational model as the fundamental data structuring and organization formalism, and SQL as the primary data access language. The current technology and commercial products have been developed with the primary purpose of supporting data processing applications. These applications typically manipulate collections of relatively simple data whose interrelationships can be modeled in a relatively straightforward manner. Additionally, the access to data can be supported by well-defined primitive operators.

The data and information that is manipulated by knowledge base systems are more complex, with complex relationships among them. Furthermore, as the development of languages such as Datalog [CGT89] demonstrate, their manipulation requires operators more complex and powerful than relational calculus and algebra. Object-orientation is expected to play a role in the development of KBMSs both as a system structuring paradigm and as a data management system.

Even though there have been a number of early efforts [Fis87, LRV88, MSOP86, SS90, SZ90], there is no commonly accepted object data model formalization. Each model differs in its formalism, support for features such as object identity [KC86], encapsulation of state and behavior [SB85], type inheritance [CW85] and typed collections.

The features that an object-oriented database management system (OODBMS) should provide is a matter of some controversy (see, for example, [ABD+89] and [SRL+90]). However, to be successful, OODBMSs should at least provide the functionality of relational systems. These

*Current address: Banyan Systems, Inc., 115 Flanders Road, Westboro, MA 01581.

functions include a declarative query language, transactions, view management and so on. Our work, and this chapter, concentrate on one of these issues: the design of query languages, their formalization, and their processing in OODBMSs. We recognize that it may be too early for an exhaustive discussion of these issues. Work on query models and query processing in OODBMSs is quite recent and there are undoubtedly many other relevant issues that have yet to be uncovered. We, therefore, restrict our discussion to those issues that have been addressed in our own research [SÖ90c, SÖ90d, SÖ91]. We also rely heavily on [YO91] which provides a framework for evaluating query models, specifically object algebras.

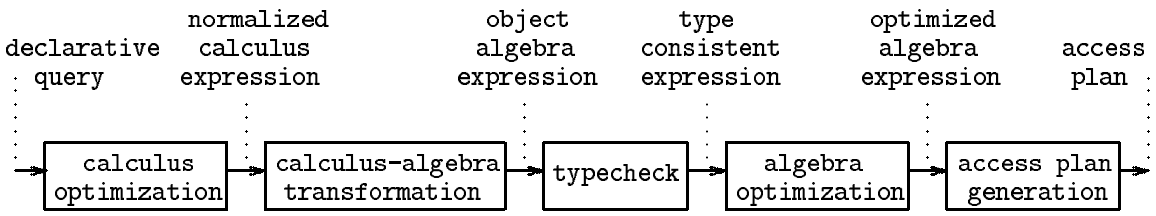

Figure 1: Query processing methodology

We extend the relational query processing methodology [JK84, GV89] as depicted in Figure 1. The steps of the methodology are as follows. Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first reduced to a normalized form by eliminating duplicates, applying identities and rewriting. The normalized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the extents of types in the database or user defined collections of objects. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested function. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence preserving rewrite rules to the type consistent algebra expression. Lastly, an execution plan which takes into account object implementations is generated from the optimized algebra expression.

The above methodology forms the framework of this chapter. Our presentation in this chapter follows two parallel tracks. We discuss the fundamental design and modeling issues, and also demonstrate how these issues can be addressed by referring to previous work [SÖ90c, SÖ90d, Str91] as an example. We first take a detour in the next section to discuss the nature of knowledge base systems and the role of object-orientation in the design of KBMSs. In Section 3 we address object-oriented data model issues and how they relate to query models. In Section 4, we discuss specifics of query model issues including the definition of a calculus and an algebra. In this chapter we distinguish between a data model and a query model. This separation is useful in OODBMSs with their rich data structuring and type systems. A data model, in this context, defines the logical structuring of the objects while the query model concerns the procedural and declarative access primitives, their safety, equivalence, completeness, and so on. Type checking and type inference rules are covered in Section 5. We also define the typechecking rules for our object algebra in this section. Optimization of algebraic expressions, covering query rewrite rules is the topic of Section 6. Section 7 addresses the final step of the methodology including object manager design issues and the generation of execution plans by mapping algebraic expressions to object manager operations. Our work on this subject [SÖ91] does not address full optimization of object-oriented queries, but

only the generation of alternative execution plans. Given a cost function that can be used to compute the cost of each plan, the optimal plan among the alternatives can be chosen. This is an area that we have not yet fully studied. This and other related and important problems that we have not addressed in our work are briefly reviewed in Section 8. Finally, in Section 9, we provide some concluding remarks on the suitability of the methodology and the remaining open problems. We assume the reader has some familiarity with object-oriented concepts and terminology.

2 Knowledge Base Systems

In this section we discuss the nature of knowledge base systems and then cover some arguments in favor of using the object-oriented paradigm to organize KBMSs.

2.1 Nature of Knowledge Base Systems

What is a knowledge base and how does it differ from a database or an expert system? This question is the subject of considerable debate within the research community without much of a consensus. In fact, there are many instances in AI and database literature where "knowledge base system" is used synonymously with "expert system." It is not our intention to engage in a detailed discussion of the nature of knowledge base systems in this section. There have been a number of attempts to address specifically this issue (e.g., [Fro86, Ull88, Wie84, Wie86]). In [Özs89] we give the following working definition which we will adopt here as well. A *knowledge base* is a structured collection of

- *data* representing facts about some aspects of a domain of discourse that is being modeled (sometimes called the *extensional database* or *fact base*), and
- *knowledge* that represents a higher level of interpretation and understanding of that domain of discourse (sometimes called the *intensional database* or *rule base*).

Thus, a knowledge base can be considered as the composition of the intensional and the extensional databases. A *knowledge base management system* can be defined as a tool which provides

- facilities for managing the intensional database as well as the extensional database,
- a language facility that enables access to the knowledge base, and
- mechanisms for the application of the knowledge to the data in order to respond to queries that require reasoning about the facts.

The language facility assists in accessing the database in the traditional sense as well as enabling the issuing of queries that require some reasoning on the factual data. The results that are returned are basically new knowledge about the application domain.

Let us consider an example from the office information systems domain, specifically a hypertext system. We will be using this example throughout this chapter. The hypertext system stores, among other things, information about documents, their authors, etc. In such a system the assertion "Document X is authored by Joe Smith and is about cruise missiles" is a fact which can be stored in a traditional database. The DBMS can then answer queries of the form "Who is the author of document X?" or "List all the documents about cruise missiles?" However, the statement "Documents about cruise missiles are considered top secret" is considered knowledge and is difficult to store and manage in a traditional database. For example, the DBMS would not be able to easily

handle the query "Is document X top secret?" Responding to this query goes beyond mere retrieval of stored facts and requires *reasoning* capabilities[1].

A knowledge base would store the fact "Document X is authored by Joe Smith and is about cruise missiles" in its extensional database and the knowledge "Documents about cruise missiles are considered top secret" in its intensional database. Then the query "Is document X top secret?" can be processed by the KBMS using its reasoning capabilities to produce the answer "Yes." The clear separation of the intensional and the extensional databases is, of course, the more difficult problem. We avoid that discussion in this chapter and, instead, refer the reader to [Wie86, Fro86].

2.2 Object-Oriented KBMS Organization

The architecture of a KBMS that provides the features mentioned above has been discussed for some time. There have been attempts at coupling expert systems with a traditional DBMS. In such designs, the extensional database is maintained by a traditional DBMS [AW86]. The expert system issues calls to the DBMS to access this data. Such an approach, which is sometimes called *loose coupling* [SH88] is depicted in Figure 2. There is an important problem with this approach. Only the fact base (extensional database) is managed by means of a DBMS; the knowledge is implicit and embedded in the expert system code. Thus, the management of knowledge follows a pattern which is identical to the management of data in traditional file processing: the knowledge storage and processing is embedded in the application code rather than being abstracted out. Explicit storing of knowledge as well as data and its management by a generalized tool (Figure 3) brings the advantages of data management to the AI application that is developed on top of a KBMS.

A tighter integration between the intensional and extensional databases and the reasoning capability leads to *knowledge representation independence* [Bro89] and provides a first order differentiation betwen KBMSs and expert systems:

1. A KBMS can reason about the stored facts to produce an answer to posed queries; that is where its functionality ends. However, an expert system can take this response and use it to solve a problem.

   Consider the hypertext example. When the KBMS responds to the query "Is document X top secret?" by a "Yes" answer, it has successfully completed its task. An expert system, on the other hand, takes that response and uses it to reach a decision and take some action (such as deciding on a security and access control action). To demonstrate this point, consider the more complicated knowledge base query "Can Officer X read document Y?" The expert system uses the reasoning capability of the KBMS to determine the security level of document Y and the security clearance of Sergeant X. It has domain specific knowledge to link X's security clearance with Y's security level to determine the outcome.

2. As a follow-up of the above point, an expert system is application specific. It knows how to use a set of facts and knowledge (inference rules) in order to reach a solution of a problem. A KBMS, on the other hand, is quite general, even in its application of inference rules. It simply stores, manages and provides access to a set of facts and applies inference rules to produce knowledge without any reference to the use of this knowledge in a given problem domain. In fact, a KBMS may manage facts and knowledge for a number of expert systems. In both of these cases inference rules that are stored in the intensional database may come from one application domain (e.g., medical diagnosis), however, the KBMS does not know

[1] We assume that the knowledge is associated with a class of documents. Traditional database systems can model this situation (inefficiently) by associating the secrecy knowledge with individual documents.

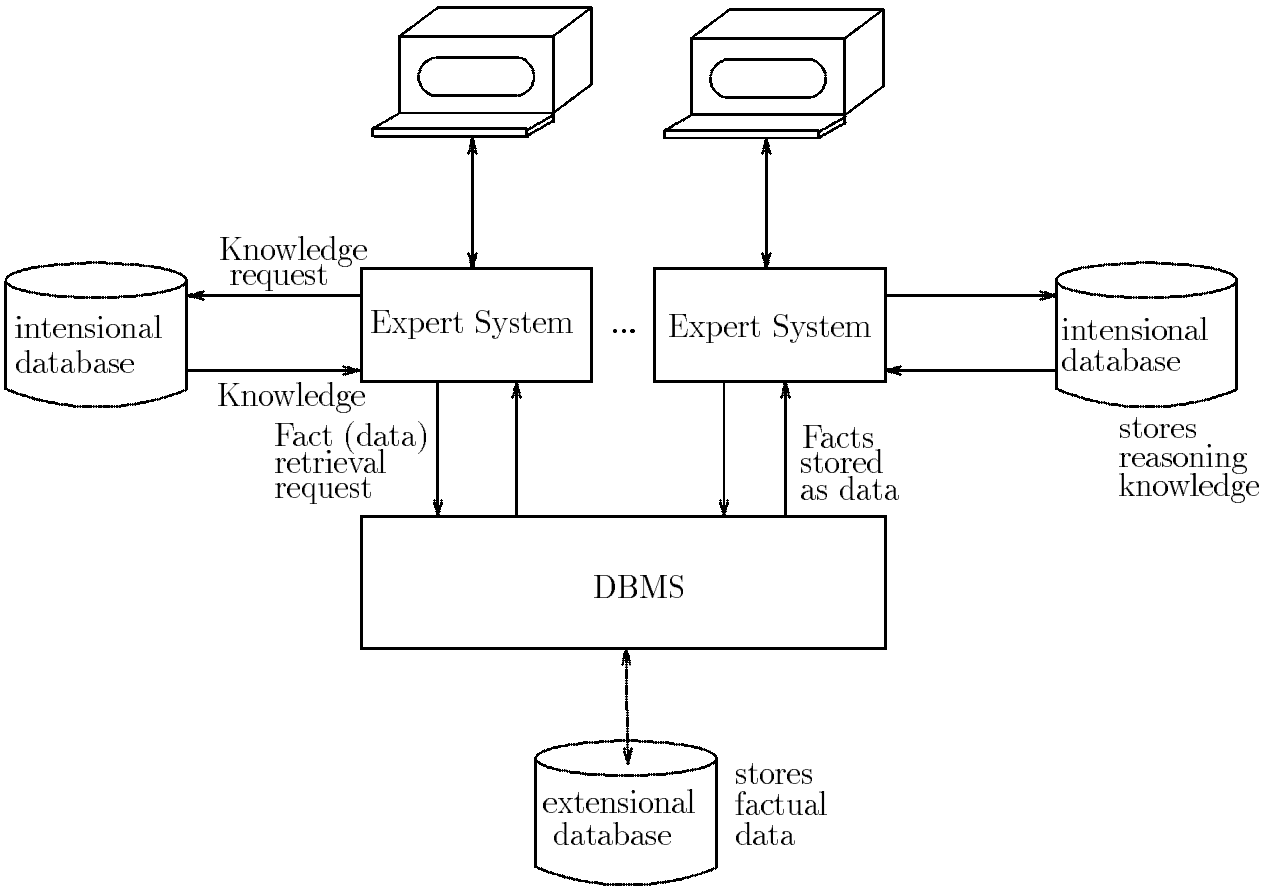

Figure 2: Loose Coupling Between Expert Systems and DBMS

the semantics of these rules in that domain and, therefore, does not know how to use them in solving a domain specific problem.

Viewed in this fashion, the KBMS constitutes the technological base for expert systems just as the DBMS forms the technological base for data processing applications.

What is the role of object-orientation in this general framework? One way to answer this question is to point out that frame-based [Min75] AI applications — for which there are many examples — embody a number of features of object-oriented systems. Along the same lines, one can cite the many object-oriented AI programming languages such as Flavors [Moo86], CommonLoops [BKK+86], and CLOS [Moo89]. However this is repeating the obvious without clarifying the role of the object-oriented technology in AI applications. Curiously, it is not easy to find compelling and constructive arguments in favor of object-oriented technology as a structuring paradigm in the AI literature. The systems built using object-oriented tools and languages exist, but the conceptual generalizations regarding the conditions under which the technology should be used and how it should be used are harder to find.

An argument in favor of object-oriented technology can be made in terms of the nature of the intensional database that is managed by a KBMS. In the previous section we indicated that AI applications deal with complex data with complex relationships. Keeping with the terminology introduced in this section, the reference is to the representation of knowledge in the intensional

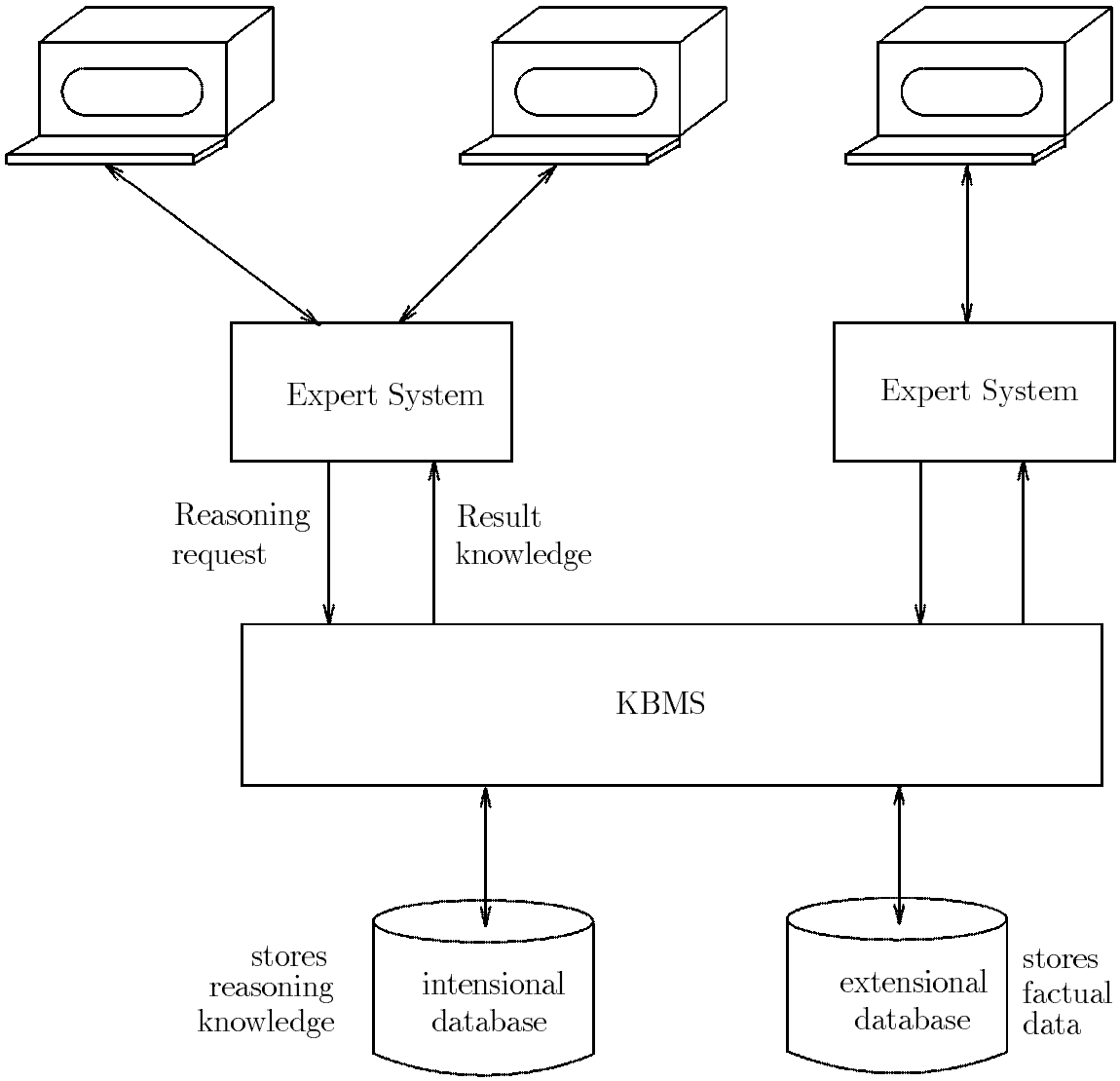

Figure 3: Knowledge Base Approach

database. These features of the intensional database structure, namely "complex and widely-varying data types" [Man89a] and the complex interrelationships among these data types appear particularly suitable for the application of object-oriented technology. As quoted in [Man89a], the following discussion from [Feu89] justifies these claims:

> "The criterion is 'where is the important information?' Is it in numbers and values, or are the relationships between the things in your database what you care about? Is it the structure or is it the data? If the important information is in the data, then you can use a relational database, but if the knowledge that you care about is how things are structured and how they are related, then you probably want an OODB ... If you just want an accounting of planes and ships and how much they cost, you can use a relational database. But if you want to know how parts and subassemblies relate to each other in an airplane or a ship, then you should use an OODB ... The place to start is where they are needed the most, where structure and modeling design are the

driving force."

Along the same lines, Sheu et al [SKY89] indicate that the "framework of object-oriented knowledge base is significantly different from most [non-object-oriented] knowledge base frameworks ... in the following aspects:

1. Since it [i.e., object-oriented knowledge base] deals with objects explicitly, more semantics can be associated to different entities.

2. It treats procedures as first-class entities, and therefore our knowledge about procedures can be included and used."

We should note that the inferencing function of a KBMS may be satisfied and enhanced by the inclusion of active objects in the model [DBM88]. Active objects observe events in the system and react to them by triggering certain actions. The events that are to be monitored, the conditions that have to be fulfilled, and the actins that are executed in response are typically defined in the form of event-condition-action (ECA) rules.

In addition to its potential use as a representation formalism for the intensional database, object-oriented technology can be used as an architectural paradigm for organizing the KBMS itself. The intensional and extensional databases may have different representation, access, and performance requirements. The object-oriented technology is an appealing candidate to deal with this heterogeneity because it allows the encapsulation of the two databases and their management routines

while, at the same time, providing a more uniform interface to the outside world by means of the abstraction capabilities. Manola suggests that the next generation knowledge-based information systems will involve "integration of heterogeneous information sources, including heterogeneous distributed databases, knowledge-based systems (such as expert systems) involving heterogeneous knowledge representations, and conventional programs and their associated processors." [Man89b]. The object-oriented technology can be a suitable vehicle for this "integration" and for the uniform management of "heterogeneity."

Assuming that the case for the object-oriented technology in knowledge base systems is successfully made, we can now concentrate on the design issues in providing a declarative query facility for object-oriented database systems.

3 Object Data Model Issues

The power and flexibility of object-oriented systems introduce considerable complexity into their models. Even the feasibility of defining an "object data model" in the same sense as the relational model is questioned [Mai89]. A comprehensive discussion and treatment of all these issues is beyond our scope, which is restricted to those data model issues that relate directly to the definition of query languages and their processing.

3.1 Model Design Considerations

The following aspects of the data model have a direct bearing on the query model and the capabilities that need to be included in a query processor. These issues are not entirely independent of each other. We illustrate how one design decision can affect others.

Nature of an "object". There are different definitions of an "object". Some object data models consider objects simply as complex data structures [BK86, LRV88, Osb88], somewhat similar to the nested relation models that permit relation-valued attributes. This approach is common to those models that are developed to deal with complex object structures as they exist, for example, in engineering applications. Other object data models consider objects to be instances of *abstract data types* (ADTs) [ACO85, SZ90], which encapsulate the representation of the objects together with a set of public methods that can be used to access them. In this case, the type is a template for its instances.

The variation in the level of encapsulation enforced by data models affects query models in the following sense. The query model must fully describe the visible components of objects which can be accessed by query primitives. For example, if objects are tuple-valued as in [BKK88], then query expressions can directly access tuple fields by name. Furthermore, the allowable query primitives are dependent upon this decision. In principle, maintaining the data abstraction paradigm would require querying the database based on object behaviors, not their structure. Complete encapsulation, therefore, would require that the comparison operators in the query language be based only on identity ("are two objects the same?") not on structure. There are query models that provide a relaxed form of encapsulation by enabling some sort of structure-dependent equality check.

Relationship between objects and types. Two types of relationships have been defined between types and objects [MZO89, MB90]: *conforms-to* (cT) and *has-type* (hT). Maier et al. [MZO89] define them structurally (i.e., based on the structure of objects) while Manola and Buchmann [MB90] define the concepts behaviorally (i.e., according to the behavior of objects as defined by their methods). An object that *conforms-to* a type indicates that the object follows the (structural or behavioral) specification of that type. The *conforms-to* relation binds the object structure or behavior from below: the object minimally has the same structure or behavior as the type it conforms to, but may have additional structure or behavior.

The stronger relation *has-type* specifies that an object is *explicitly declared* to be an instance of a specific type. The set of instances of a type t_i is called its *extent* which we will denote by $ext(t_i)$. For an object o and type t

$$o\ hT\ t \implies o\ cT\ t$$

Without loss of generality, let us assume the existence of one object o and two types t_1 and t_2. Let us further assume that there has to be at least one *has-type* relationship defined for all objects. The following relationships exist between o and t_1 and t_2:

1. $o\ hT\ t_1$ and $o\ \neg cT\ t_2$
2. $o\ \neg cT\ t_1$ and $o\ hT\ t_2$
3. $o\ hT\ t_1$ and $o\ cT\ t_2$
4. $o\ cT\ t_1$ and $o\ hT\ t_2$
5. $o\ hT\ t_1$ and $o\ hT\ t_2$

The first two cases are straightforward; they simply indicate that o has been declared to be in the extent of one type and has no relation to the other. The third and fourth cases indicate that objects can conform to more than one type, but are explicitly declared to to be in the extent of only one type. In most object data models, for these two cases to occur, there has to be a

relationship between types t_1 and t_2. Similar to those between an object and a type, there are two kinds of relationships between types. Type t_1 is said to *specialize* type t_2 if the structure (or the behavior) of t_2 is included in the structure (or behavior) of t_1. The stronger relationship between t_1 and t_2 is *subtype*. Type t_1 is said to be a *subtype* of type t_2 if t_1 is explicitly declared to specialize t_2 (i.e., it is explicitly declared that the structure (or behavior) of t_2 is included in the structure (or behavior) of t_1). t_2, in this case, is called the *supertype* of t_1. Subtyping establishes an "IS-A" relationship between t_1 and t_2: "t_1 is a t_2." Thus, $o\ hT\ t_1 \implies o\ hT\ t_2$. Since, $o\ hT\ t_2 \implies o\ cT\ t_2$, it is the case that $o\ hT\ t_1 \implies o\ cT\ t_2$. Therefore, cases (3) and (4) follow from subtype/supertype relationships between t_1 and t_2. The subtype/supertype relationships between types form a hierarchy (or a semilattice, if types are allowed to have multiple supertypes) such that a parent type in the lattice is a supertype of all its children types.

In most object data models, the fifth kind of relationship can occur only if there is a subtype/supertype relationship between t_1 and t_2 (in either direction). In other words, an object o which *has-type* t_1 also *has-type* t_2 if t_1 is a subtype of t_2. In these models, each object is explicitly declared to have one and only one type and the other *has-type* relationships are those that are derived from the type semilattice. Such models are said to enforce *strong typing*.

The relevance of these relationships, especially that of subtyping, to query processing is the following. Since t_1 *subtype* $t_2 \implies [o\ hT\ t_1 \implies o\ hT\ t_2]$ and since the *has-type* relationship defines that an object is in the extent of a type, there is a relationship between the extents of t_1 and t_2 ($ext(t_1)$ and $ext(t_2)$, respectively). Specifically, $ext(t_1) \subseteq ext(t_2)$ (Figure 4). Therefore, the net effect of a query that asks for instances of t_2 is usually to retrieve instances of t_1 as well. This can be extended to multiple levels of subtype/supertype semilattice. Thus, the query result consists of the union of all objects that are instances of all types in the subtree of the type lattice rooted at the type at which the query is posed. We call this the *deep extent* of a type in [SÖ90c] and denote it $ext^*(t_i)$.

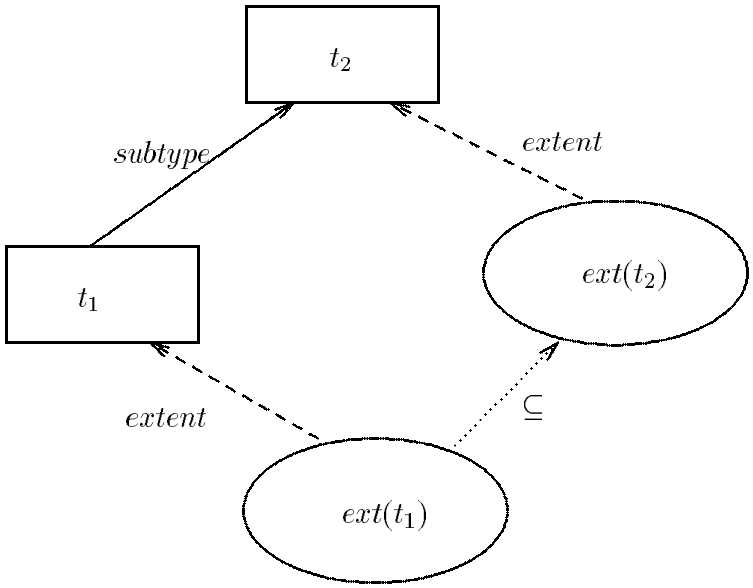

Figure 4: Subtyping Relationship

As we will discuss in the next section, there are other semantics that can be associated with the fifth kind of relationship; what we have discussed above is the more restricted and straightforward semantics. Even with this restriction, there are design alternatives that may be introduced. Some

object data models permit *variant* objects that may deviate from the template defined by the type in some manner. Others, called *prototypical object models*, "clone" objects from other existing objects rather than using a type definition as the template for object creation. These models facilitate the definition of one-of-a-kind objects that act as their own type specifications.

In systems that allow variant and prototypical objects (e.g., [Lie86, US87, MB90]), it is no longer possible to specify the full behavior of each object based on its type and this influences the kinds of optimizations that can be performed. Furthermore, the definition of the "schema" when prototypical and variant objects are supported needs to be clearly worked out. When variant objects are allowed in the data model, the type is no longer a template for its instances as we claimed before; it is only a **minimal** template. Therefore, it only defines the minimal behavior of the objects that are in the extent of that type. This is important for the following reason. If the schema defines only the minimal behavior, and a query "takes as input a schema (and a database) and generates as output another schema (and another database)" [BK90], how can the additional behavior specified by variant and prototypical objects be queried?

Single versus multiple types. As indicated above, the interpretation of the fifth type of relationship with respect to subtyping is the more restricted semantics that can be attached to it. An alternative semantics may be that objects are indeed allowed to enter into *has-type* relationships with types that have no subtype/supertype relationship with each other. In this case the object has the structure or behavior (depending upon whether the model is structural or behavioral) of two types. Consider an example of two types, `Employee` and `Student`. In this schema, a Teaching Associate can be represented as having a *has-type* relationship with both the `Employee` and the `Student` types(Figure 5(a)).

This example can be modeled in systems that enforce the more restricted semantics of the multiple *has-type* relationships by making use of the type semilattice. To model a Teaching Associate as both an employee and a student, a `TeachingAssociate` type can be created with two supertypes: `Employee` and `Student` (Figure 5(b)).

The more relaxed semantics of multiple *has-type* relationships may be considered a more flexible and, in some sense, more natural representation of the real world, but it is also more difficult to handle in a query model. The difference between a data model that allows objects to belong to multiple types and one which "simulates" the same effect by creating subtypes is subtle. In the latter, there is a new type in the type lattice which becomes part of the schema; in the former, no such type definition exists in the schema. If we assume the existence of a system-defined `MyType` function that maps an object to its type(s), in the example that we are considering, the result of applying this function to an object which represents a Teaching Associate would be different in the two cases. `MyType` would return {`Employee, Student`} if objects can belong to multiple types, but its result would be {`TeachingAssociate`} if explicit subtypes are created.

This semantics of multiple *has-type* relationships creates problems for query processing. As a direct consequence of the definition of *has-type*, a Teaching Associate which is defined to have both the type `Employee` and the type `Student` would be in the extent of both of these types. In the more restricted interpretation involving `TeachingAssociate` as a subtype of `Employee` and `Student`, on the other hand, it would be in the extent of `TeachingAssociate` and in the deep extent of `Employee` and `Student`. Those queries which ask for instances of `Student` (or of `Employee`) would retrieve the same set of instances in both cases. However, a query that asks, for example, for Teaching Associates who are in the computer science department would be easier and more efficient to process if the `TeachingAssociate` type existed. In this case, the query involves simple selection over the extent of `TeachingAssociate` (Figure 5b). In the other case (Figure 5a), however, it is necessary to do selection over the extents of both `Employee` and `Student` and then take an intersection of the two

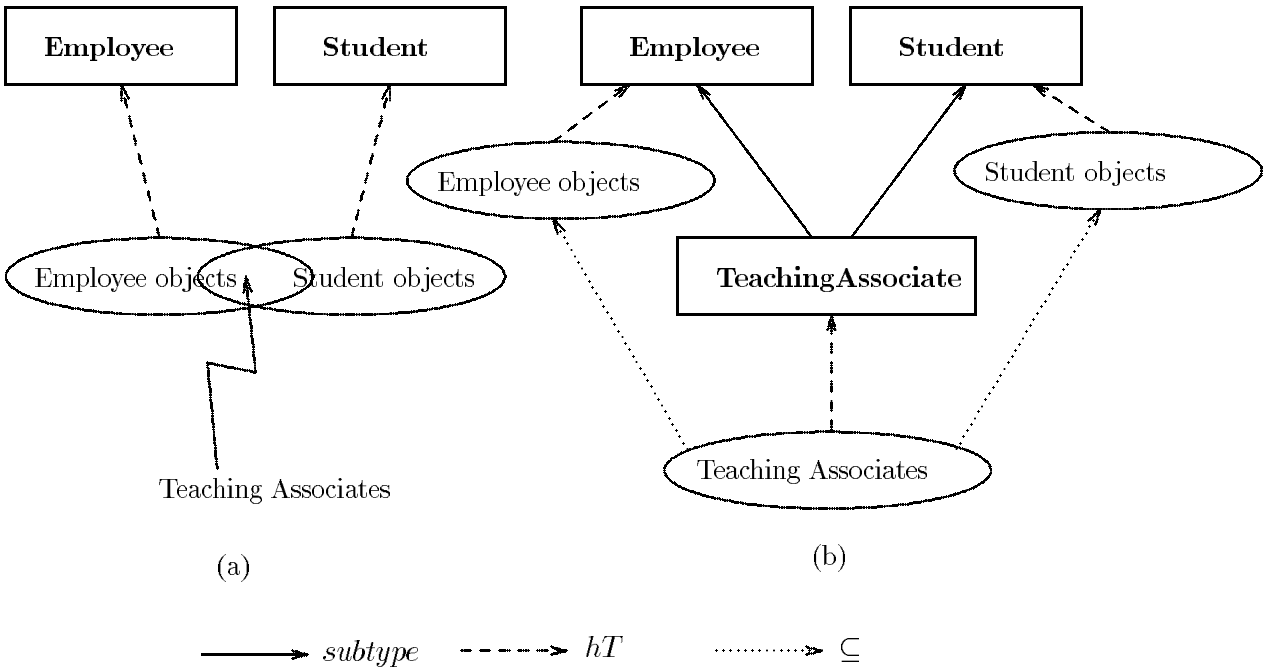

Figure 5: Alternative Representations of Multiple *has-type* Relationship

sets. This is assuming that it is only possible to serve as Teaching Associate in the department where one is a student. If this is not the case, the evaluation of the query requires some sort of "join" between, or an intersection of, the extents of the two types followed by a selection. In either case, the evaluation is expensive.

There are some data models that allow objects to have *has-type* relationships to multiple types [MZO89, MB90]. The definitions of these models do not make it clear which semantics they associate with this relationship, however. For example, TEDM [MZO89] states that t_1 *subtype* t_2 if for any object o, $o\ hT\ t_1 \implies o\ hT\ t_2$[2]. However, there is nothing in the model specification which would force the implication $o\ hT\ t_1 \implies o\ hT\ t_2$ to hold. Therefore, it is possible in TEDM for an object to be related to two types t_1 and t_2 which do not have subtype/supertype relationship with each other. Thus, the issues that we raise above with respect to query processing hold for TEDM and for FROOM [MB90] which accepts the behavioral versions of TEDM's definitions.

Classes versus collections. In the previous section we referred to the extents of types. In some object data models the concept of a type and its extent is not separated. These models typically refer to a *class* as both a type definition and as the collection of all objects that are instances of that type. Other data models separate the concepts of a *type* and a *class* as an extent of that type. Our discussion in the previous sections assume the latter.

Both of these, however, make use of the class concept as the only aggregation of instances of the associated type. A further distinction can be made between these and other models in which there is no explicit notion of a class. In these models, objects are grouped into arbitrary collections.

As far as object creation is concerned, the difference is the following. In models with the class

[2]Note that TEDM defines subtyping in terms of the *has-type* relationship. Our definitions in the previous section are in the opposite direction: subtyping implies *has-type* relationship between extents. Both ways of writing it is correct. Combining the two would allow a stronger definition of subtyping by replacing "if" with "if and only if".

concept, an object automatically becomes a member of the class that represents the extent of the associated type when it is created. In those that utilize user-defined collections as aggregations of objects, the user is expected to explicitly specify the membership of the object in one or more of the collections.

From the perspective of query models, the difference is with respect to the targets of queries. In class-based models, the queries are posed on classes. In models based on user-defined collections, queries are specified on any one (or more) of the collections. Consider the hypertext example and assume the existence of a `Document` type. In the first case, there is a `Document` class over which queries are defined. Thus, a query which asks for authors of documents that satisfy a selection predicate will retrieve **all** documents that satisfy that predicate. In the second case, however, no `Document` class exists. Maybe there are explicit collections such as `SecretDocuments, TopSecretDocuments, OpenDocuments` and the query either has to be defined over one of them (resulting in the retrieval of only the documents in that collection) or the user has to explicitly specify the query on all three collections and union the results. Note that this distinction is important in the context of defining what the schema is and, therefore, what the target of a query is.

It has been pointed out that object data models that employ "class as the extent of type" approach make it easier to conceptualize a query [Kim89]. Furthermore, it is possible in these models to exploit the subtype/supertype relationship [YO91] through deep extents. In other words, there is a clear semantics of retrieving objects that are in the deep extent of a type in these models whereas this cannot be said for collection-based models.

Recently, the value of combining both approaches rather than viewing them as alternatives has been recognized. Beeri [Bee90] has proposed a model which defines classes as extents of types with automatic membership semantics and collections as user-defined subsets of classes. In his model, a collection has to be a subset of one class, resulting in homogeneous collections. In [PÖ91], we adopt the same approach and separate classes (as extents of types) from user-defined collections while supporting both concepts. In contrast to Beeri, however, we permit heterogeneous collections to facilitate uniform treatment of results of queries which may target multiple classes (or collections).

Mechanism for sharing. One of the strengths of object-oriented models is that they provide mechanisms for sharing among objects. Two types of sharing are possible: sharing of implementation and sharing of behavior. Behavioral sharing is what we called subtype/supertype relationship above. It is important to differentiate this from implementation sharing. As noted by Nierstrasz [Nie89], "many of the 'problems' with inheritance arise from the discrepancy between these two notions." He goes further and associates subtyping with types, and inheritance with classes. This follows from the model that he describes where a type is defined as an abstract specification of the behavior of objects of that type, and a class as a specification of the template for the implementation of these objects (in addition to serving as a collection of all the objects of that type). This association may be valid especially in models that support "classes as the extent of type" approach, however it is not the critical point. The important aspect is to note that the relationship between type definitions and the relationship between implementations can be (and, some claim, should be) separated. More traditional object-oriented languages such as Smalltalk [GR83] bundle these two concepts.

There have been two proposals for implementing sharing, one based on *inheritance* (e.g., Smalltalk) and the other based on *delegation* [Lie86]. In inheritance, the sharing is based on a semilattice of types or classes (assuming Nierstrasz's model where classes specify implementation templates). Thus, type t_1 of Figure 4 which is a child of type t_2 in the hierarchy, inherits behav-

ior from t_2[3]. In delegation, sharing is achieved by an object explicitly delegating its behavior or implementation to another object. By and large, the choice of the sharing mechanism has been tied to the choice of the type system. Those models which allow variant and prototypical objects typically implement delegation as the sharing mechanism, while those that enforce strong typing implement inheritance. There has been some recognition that the two mechanisms are similar [SLU89, Ste87], but the semantics of an object data model that permits prototypical and variant objects, but implements inheritance as the sharing mechanism is likely to be quite complicated. The specification of inheritance between types and the inheritance between instances of those types (which are different due to the existence of variant objects) and the link between the two is bound to be quite complex. These complications affect query processing directly since the optimization of queries in an object-oriented system can and should take advantage of the semantics of sharing.

Uniformity of the model. Is everything in the system an object? Some data models (e.g., OODAPLEX [Day89], FUGUE [HZ88], FROOM [MB90]) treat as objects types, methods, and anything else that can be defined in the system. Such models bring uniformity to the treatment of objects. This is in contrast to other models where concepts such as methods and types are treated as meta-information separate from objects.

Uniformity in the object data model affects the query formalism in various ways. First, the notion of a "schema" as a source of supplemental meta-information, separate from the database being queried, is replaced by the concept of taking the schema and including it as part of the database itself. In addition to collapsing the potential hierarchy of meta, meta-meta, etc. information into a single level, self-describing system, this provides greater flexibility by allowing the same efficient techniques specified in the query formalism to be used on what was previously considered meta-information. An advantage of this approach is that one may now apply the query formalism to the schema data in order to extract semantic information on the objects being queried and manipulation of the schema through the query formalism may be possible as well. A second affect of uniformity on the query formalism refers to the way in which *method objects* are handled. If methods are indeed objects, then the query model should have the ability to cope with them. (i.e. invoke them, access their bodies, etc.). As observed in [MB90], such capabilities require query language facilities for invoking methods, passing parameters to methods and dealing with returned results.

A final consideration of uniformity deals with that of the query formalism itself. In order to have a truely "uniform" system, shouldn't the query model become part of the object space as well? If this were the case, then one could envision even greater querying power and flexibility by allowing queries on the query model itself. This may be useful, for example, in asking the database questions on how a certain query would be, or was, processed. This could, in a sense, serve as an explanation facility for the system.

The introduction of uniformity to the data model increases the power and flexibilty of the query formalism, but at the cost of additional complexity in its specification and formalisation. In some sense, all of the discussions of the previous sections have to be revisited if complete uniformity of the model is assumed.

3.2 A Sample Object Data Model

In this section we describe the object data model that forms the basis of our investigation into query processing in object-oriented databases. We only summarize the model here and refer to

[3]The counterpart holds for classes with respect to implementation sharing.

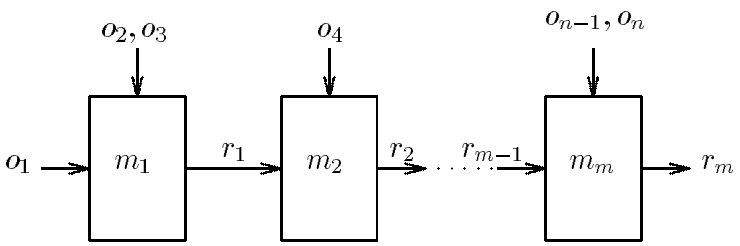

Figure 6: Composition of method applications.

[SÖ90c] for a full and formal definition. The identifying characteristics of our data model are the following. Objects are viewed as instances of abstract data types which can only be manipulated via functions defined by the type. Types are organized in an inheritance hierarchy which allows multiple inheritance. Each object has a unique, time invariant identity which is independent of its state. Relations on object identities such as equality and set inclusion provide the basis for query primitives which qualify query operators. All other relations among objects are implemented by the ADT interfaces. We briefly highlight these properties below.

3.2.1 Classes and Methods

In our model, we do not make a distinction between a class and a type. Therefore, a *class* defines both a type interface via *methods* and stands for all the objects which are instances of the type. Methods are named functions whose arguments and result are objects. Each method has a signature of the form $C_1 \times \ldots \times C_n \rightarrow C_{result}$ where $C_1 \ldots C_n$ specify the class of the argument objects and C_{result} specifies the class of the result object. All classes in the database form a class lattice where the root node represents the most general class of objects and any individual class may have multiple parents. Subclasses inherit behavior from their parents and may define additional methods. Thus, the class lattice provides inclusion polymorphism [CW85] which allows an object of class C to be used in any context specifying a superclass of C [SZ90]. This is similar to the conformance relationship of Emerald [BHJ+87] and the subtype relationship in the functional language Ponder [Fai88].

3.2.2 Object Behavior

Objects encapsulate a state and a behavior. Methods, defined on the class which an object is an instance of, define the object's behavior. Behavior is revealed by applying a method to an object. The result of a method application is another object. The dot notation $<o_1 \ldots o_n>.m_1.m_2 \cdots m_m$ is used to denote method application and method composition. Assuming methods m_1 and m_m take three arguments each, and method m_2 takes 2 arguments, then Figure 6 illustrates the processing denoted by this operation. Method m_1 is applied to objects $<o_1, o_2, o_3>$ resulting in object r_1, method m_2 is applied to objects $<r_1, o_4>$ returning object r_2, and so on until the final result object r_m is obtained by applying method m_m to objects $<r_{m-1}, o_{n-1}, o_n>$. Note that the dot notation denotes function application and composition, not the traditional record field selection (attribute selection) as in [BKK88, Car84, MS86]. $<o_1 \ldots o_n>.mlist$ will be used when the list of method names is unimportant.

3.2.3 Object State

An object's state is captured by its value which is distinct from its identity [KC86, SB85]. Object values are either an *atomic value* provided by the database system (int, string, uninterpreted byte sequence [CDRS86]), a *set value* which is a collection of object identifiers, or a *structural value*. Structural values are visible only to class implementors and can encompass attributes (tuples), discriminated unions, etc. as in [ACO85]. Any aspects of structural values which are required by users of a class should be revealed by the implementor via a method (ultimately delivering either an atomic or set value).

3.3 Example Database

The hypertext application is selected as an example because it belongs to an application domain (office information systems) that is claimed to potentially benefit from the object-oriented database technology. Specifically, a hypertext system requires persistent data, has a large number of data types and many types of ad hoc queries can be posed. The basic underlying concept is a simple one. Windows on the screen are associated with units of information stored in a database [Con87]. Information units are related to one another via *links*. Links are typed in the sense that some may be used to specify the structural composition of a document (structural links), some may point to related information which supports the primary theme (referential links), and some may point to comments made by reviewers (note links). Users of the hypertext system browse through documents by traversing links and examining nodes of interest. This approach is a powerful communications tool as documents do not need to be structured linearly and users can sidetrack to follow related trails of information in whatever order they desire.

Information units are referred to as *nodes* and can encompass text, graphics, computer generated sound, and even executable programs. The example will be restricted to textual nodes. A *document* is a set of nodes connected by links with one node designated as the *root* node. Figure 7 depicts a hypertext system with structural links shown as solid lines and referential links shown as dotted lines. The nodes labeled **A** and **B** are root nodes. Documents can have any structure desired. Here the documents rooted at **A** and **B** are linear and hierarchical respectively. In general, there are no restrictions on links thereby allowing nodes to be a part of multiple documents, as in the case of **C**, or to exist outside of a document as in the case of nodes **D**, **E** and **F**. The forest of links associated with a document or group of documents is called a *web*.

The hypertext database can be browsed in three ways. One method is to follow links and to open windows on nodes to examine their contents[4]. Another method is to graphically display the web associated with a document and selectively examine nodes of interest. Third, the database can be queried to identify nodes meeting some criteria. Nodes are qualified using selection criteria appropriate to the node type. For example, textual nodes may be selected based upon a keyword search while graphics nodes are selected based upon pattern recognition. The query mechanism can also be used to filter the nodes and links presented to the user when viewing the web of a document. Schatz and Caplinger [SC88] note that as a hypertext system grows, its web becomes less connected. This is due to the existence of documents which do not reference one another. In this situation, link following and web display as methods for finding related units of information are of limited usefulness. As a result, the ad hoc query capabilities become more important as the hypertext system grows in size.

The design of the user interface contributes greatly to the usefulness of a hypertext system. The ease and speed with which links can be followed and windows opened on information units can

[4]Most systems implement link following and window invocation as a single mouse command.

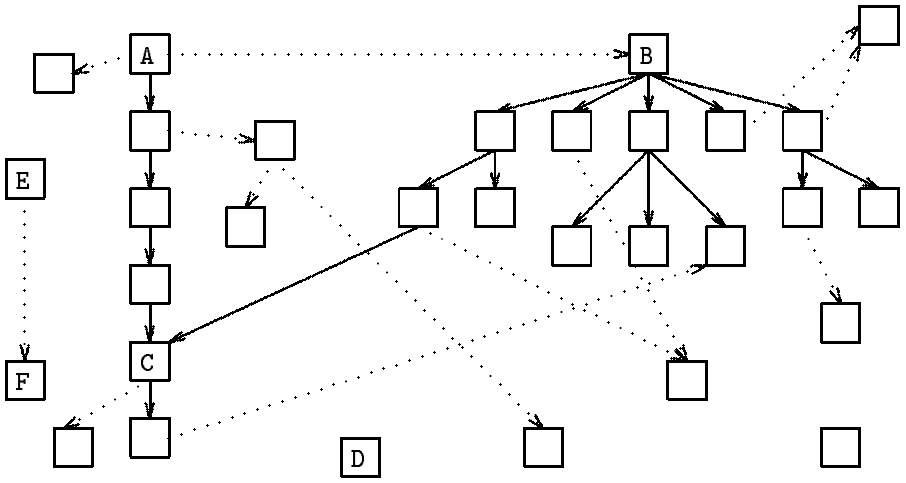

Figure 7: A *web* of hypertext nodes.

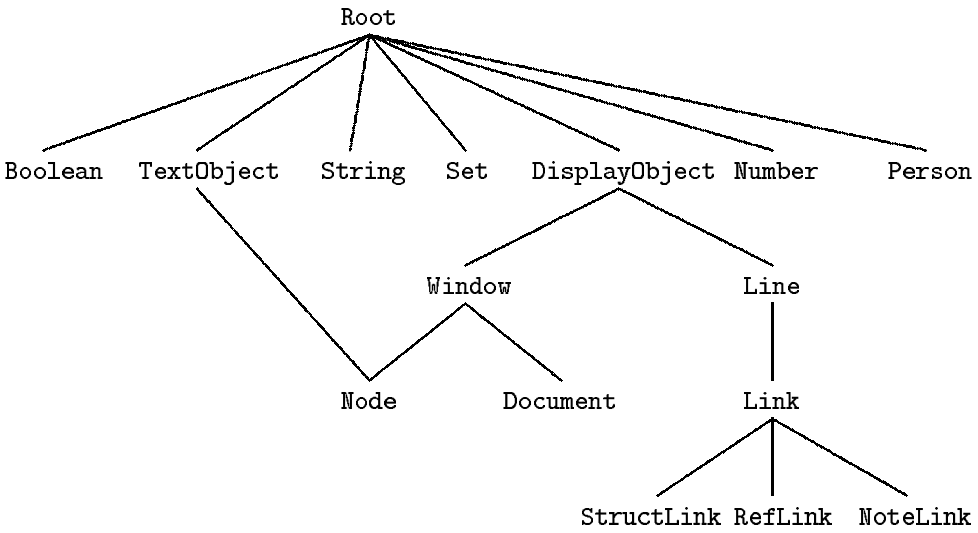

Figure 8: Classification graph for a hypertext system.

make the difference between a system which augments concurrent thought processes and one which merely stores large amounts of related data. Although implementations such as KMS [AMY88], Notecards [Hal88] and Intermedia [Con87] each have a unique user interface, a common, low level architecture can be identified. Campbell and Goodman [CG88] call this common set of features the *Hypertext Abstract Machine* (HAM) and show how several well known systems can be implemented on the standardized hypertext subsystem. The example implements a subset of the HAM using the object-oriented database model presented in this chapter.

The class lattice for the hypertext database is given in Figure 8. The classes *Boolean*, *String*, *Set*, and *Number* should be considered as being predefined by the database management system while all other classes are defined by the hypertext database implementor. The signatures of methods defined by each class are given in Table 1. The classes *Node*, *Link*, and *Document* closely reflect the logical hypertext structure described earlier. However, some implementation details are significant. Since a node may belong to several documents concurrently, the links emanating from it belong to several documents as well. The method $links : Node \times Doc \rightarrow Set$ returns the set of

Table 1: Method signatures for classes of the hypertext database.

Name	Method Signatures
Root	
Boolean (Bool)	*negate:* $Bool \rightarrow Bool$
TextObject (TO)	*contains:* $TO \times Str \rightarrow Bool$ *creator:* $TO \rightarrow Person$ *keywords:* $TO \rightarrow Set$ *edit:* $TO \rightarrow TO$
String (Str)	*contains:* $Str \times Str \rightarrow Bool$ *concat:* $Str \times Str \rightarrow Str$ *within:* $Str \times Str \times Num \rightarrow Bool$
Set	*size:* $Set \rightarrow Num$ *add:* $Set \times Root \rightarrow Set$
DisplayObject (DO)	*display:* $DO \rightarrow DO$ *isColor:* $DO \rightarrow Bool$
Number (Num)	*add:* $Num \times Num \rightarrow Num$ *greater:* $Num \times Num \rightarrow Bool$
Person	*age:* $Person \rightarrow Num$ *expertise:* $Person \rightarrow Set$ *mother:* $Person \rightarrow Person$ *father:* $Person \rightarrow Person$ *children:* $Person \times Person \rightarrow Set$
Window	
Link	*creator:* $Link \rightarrow Person$ *from:* $Link \rightarrow Node$ *to:* $Link \rightarrow Node$ *part_of:* $Link \rightarrow Doc$
Node	*links:* $Node \times Doc \rightarrow Set$
Document (Doc)	*author:* $Doc \rightarrow Person$ *co-authors:* $Doc \rightarrow Set$ *title:* $Doc \rightarrow Str$ *keywords:* $Doc \rightarrow Set$ *rootnode:* $Doc \rightarrow Node$
RefLink	
NoteLink	
StructLink	

links emanating from a node for a given document.

Links identify their source and destination node by means of the methods *from* and *to*. Unlike the HAM which tags links with an attribute, we have chosen to separate link types by defining the subclasses *StructLink*, *RefLink* and *NoteLink*. This provides the opportunity to a-priori restrict the scope of a query by defining it to range over the appropriate subclass of *Link*. *Link* is a specialization of the graphical object class *Line*.

Documents and nodes both require a display ability and thus are a subclass of *Window*. *Node* additionally inherits its text handling behavior from *TextObject* which defines methods such as *contains* for testing substring containment and *edit*. Although a document is a collection of nodes, it would be incorrect to implement *Document* as a specialization of *Node*. Instead, instances of *Document* have a structural value which captures the document structure but is hidden from users. Access to this structure is provided by methods on the *Document* class thereby preserving the ADT abstraction.

4 Query Model Issues

There are a large number of issues to consider in designing query models, many of which are still under investigation. In the first part of this section we concentrate on three key trade-offs of OODB query facilities: (1) formal *vs* ad hoc query languages, (2) predicates based upon structure *vs* behavior, and (3) object preserving *vs* object-generating operations. We also discuss a number of other query model design issues. A more detailed discussion of object algebra design considerations can be found in [YO91]. In the second part of this section we present calculus and algebra definitions for the data model presented in Section 3.2.

4.1 Design Alternatives

Formal versus ad hoc query languages. Formal query languages [Osb88, SZ90, SÖ90c] have several properties not found in ad hoc query languages [Fis87, MSOP86] making them more suitable for formal analysis. Most importantly, their semantics are well defined which simplifies formal proofs about their properties. Common types of formal query languages are a calculus or an algebra. A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in an algebra are procedural in nature but can be optimized. Algebras provide a sound foundation for rule-based transformation systems [Fre87, GD87, HFLP89] which allow experimentation with various optimization strategies. A large body of work exists on algebras for other data models (see, for example, [AB84, JS82]). Defining OODB query requirements formally in terms of an algebra facilitates comparisons with these other models.

An important aspect of formal query languages is whether or not they support a calculus definition (in the sense of the relational calculus). If declarative languages are to be provided at the user interface, there is a need to define a formal *object calculus*. We have defined such a calculus in [SÖ90c], but calculus definitions are typically lacking in object-oriented query research.

Definition of a calculus raises a number of interesting issues. The notion of *completeness* (in the same sense as relational completeness) has to be worked out, since it influences the set of algebraic operators. Completeness requires the calculus and the algebra to be equivalent. *Safety* of calculus expressions is also an issue that needs to be worked out. Safe expressions guarantee that queries retrieve a finite set of objects in finite amount of time [OW89]. Finally, efficient algorithms need to be developed to translate safe calculus expressions to algebraic ones. The work reported in [SÖ90c] defines a "restricted" calculus (therefore is only partially complete) and gives a translation algorithm to an object algebra.

Predicates based upon structure versus behavior. As discussed in the previous section, some object models implement complex objects whose internal structure is visible while others view objects as instances of abstract data types. Access to objects which are instances of an ADT is through a public interface. This interface defines the behavior of the object. Although the two views of objects appear incompatible, the ADT approach can effectively model complex objects by including *get* and *put* methods for each of the components of the internal structure [Zdo86]. Thus, a query language which supports predicates based on object behavior is more general while still allowing knowledge of object representations to be introduced in a later stage of query processing.

Object preserving versus object creating operations. A distinction can be made between object preserving and object creating query operations [SS90]. Object preserving query languages [ASL89, ACO85, MSOP86] return objects which exist in the original database. Object creating languages [Kim89, LRV88, Osb88, SZ90, Dav90, DD91] answer queries by creating new objects from components of other objects. The new objects have a unique identity and some criteria is used to appropriately establish their supertype/subtype properties. In one sense this violates the integrity afforded by objects with identity as objects with no apparent relation to each other can be combined and presented as a new object which presumes to encapsulate some well defined behavior. But the requirement for combining objects into new relationships does exist; either for output purposes or for further processing as in knowledge bases where knowledge is acquired by forming new relationships among existing facts.

Notice that any object-oriented query language must have a complete object preserving query facility independent of whether it additionally creates new objects. The ability to retrieve any object in the database utilizing relationships defined by the inheritance lattice or defined by ADT operations on objects is a fundamental requirement. The addition of object creating operations increases the power of the language, but also raises a number of issues such as the type of the created objects and the operations that they support.

Closed versus open algebras. One of the strengths of the relational algebra is that it is *closed* so that the output of one operation can become an input to the next. Extension of this concept to object algebras is considered "highly desirable" [BK90]. Closure is somewhat more complicated in OODBMSs, however. The simplifying factor in relational systems is that the operand(s) as well as the result of any algebraic operation are relations. Thus, all operators have **one type** of input and generate **one type** of output: relation. In object-oriented systems, the schema consists of many types. Thus, the closure property has to be redefined to handle the multiplicity of types. A closed object algebra consists of operators each of which operates on set(s) of objects belonging to one or more types in the type system and outputs a set of objects belonging to one or more existing types in the type system. As observed in [BK90], most object-oriented languages are "able to map structured objects into other structured objects. However, the objects returned do not necessarily belong to any of the existing [types]."

Note that the existence of object creating algebra operators, by definition, complicates closure. The provision of heterogeneous collections as outputs of queries is also difficult to reconcile with closure. The issues relate to the determination of the type of objects in the collection which we address in Section 5.

Object algebra operator set. Few object algebras have been defined formally thus far [MD86, Osb88, SZ90, SÖ90c]. There is no agreement on the set of operators or their semantics. As we indicated before, disagreements exist on whether object creating operators should be included,

what the proper level of encapsulation should be and so on. It has been suggested that object algebras should extend relational algebra [YO91], requiring the definition of project and Cartesian product operators. However, these operators, by definition, deal with components of objects, thereby violating strict encapsulation. As we indicated above, there is probably a need to include these operators in the language, but their exact relationship to encapsulation needs to be worked out.

4.2 An Example Query Model

Since the fundamental objective of our research was the query processing methodology and techniques, we defined a formal calculus and algebra which are presented in the following sections. The finiteness and safety arguments for the object calculus and algorithms for calculus-to-algebra translation are given in [SÖ90c]. As discussed in the next section, our query primitives are restricted comparisons based on object behavior. This is fundamentally due to the strict encapsulation enforced by the data model and its treatment of objects as instances of abstract data types. The same considerations have caused us to restrict the algebra to object preserving operations. The justification for this choice is twofold. First, any OODB query language must have a complete object preserving query facility independent of whether it additionaly creates new objects. The ability to retrieve any object in the database utilizing relationships defined by the type inheritance graph or defined by ADT operations on objects is a fundamental requirement. Second, as discussed above, the issues that are raised by the definition of object creating operations were not the focus of our research and we are not clear how to consistently deal with these problems at this point.

4.2.1 Query Primitives

In principle, maintaining the data abstraction paradigm would require querying the database based on object behaviors, not their values. However, the real world is both behavior and value based, thus the query language for a database modeling the real world must allow specification and comparison of values. We define four comparison operators which can be used in queries: $==$, $\in$, $=_{\{\}}$ and $=$ whose semantics are shown in Tables 2 and 3. The $==$ operator tests for object identity equality; i.e., $o_i == o_j$ evaluates to true when o_i and o_j denote the same object. The $\in$ and $=_{\{\}}$ operators apply to set valued objects and denote set value inclusion and set value equality respectively. As shown in the tables, one of the operands can denote a value if required. The last operator, $=$, can only be used to test the value of an atomic object. In order to maintain data abstraction, no primitives are provided for querying structural values. Any aspect of structural values which are required by users of an object should be made available via methods by the class implementor.

Table 2: Semantics of $o_i \theta o_j$ as a function of the object value type.

		$o_i \theta o_j$			
o_i	o_j	$==$	$=$	$\in$	$=_{\{\}}$
atomic	atomic	T/F	T/F	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	undefined
structural	atomic	T/F	undefined	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	undefined
set	atomic	T/F	undefined	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	T/F

Table 3: Semantics of $a\theta o_i$ as a function of the object value type.

		$a\theta o_i$			
a	o_i	==	=	$\in$	$=_{\{\}}$
val_1	atomic	undefined	T/F	undefined	undefined
	structural	undefined	undefined	undefined	undefined
	set	undefined	undefined	T/F	undefined
$\{val_1, \ldots, val_n\}$	atomic	undefined	undefined	undefined	undefined
	structural	undefined	undefined	undefined	undefined
	set	undefined	undefined	undefined	T/F

Atoms are the building blocks of calculus expressions and predicates for qualifying algebra operators. They represent the primitive query operations of the data model and return a boolean result. The legal atoms are as follows:

- $o_i \theta o_j$ where:
  - o_i and o_j are object variables or denote an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - θ is one of the operators ==, $\in$ or $=_{\{\}}$.
- $a\theta o_i$ where:
  - o_i is an object variable or denotes an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - a is the textual representation of an atomic value or a set of atomic values.
  - θ is one of the operators =, $\in$ or $=_{\{\}}$.

Example 4.1 Let p, q and r be object variables. Then the following are examples of legal atoms and their semantics:

1. $(p == q)$ – Are the objects denoted by p and q the same object?
2. $(p \in <q, r>.mlist)$ – Is the identifier of p contained in the set value of the object obtained by applying the methods in *mlist* to the objects $<q, r>$?
3. $(<p, q>.mlist =_{\{\}} r)$ – Is the set value of the object obtained by applying the methods in *mlist* to the objects $<p, q>$ pairwise equal to the set value of the object denoted by r?
4. ("59" $= p$) – Is the atomic value of the object denoted by p "59"?
5. ("59" $\in p$) – Does the set value of the object denoted by p include an identifier for the object whose atomic value is "59"?
6. ({"59", "61"} $=_{\{\}} <p, q, r>.mlist$) – Does the set value of the object obtained by applying the methods in *mlist* to the objects $<p, q, r>$ contain only two identifiers for objects whose atomic values are "59" and "61"? $\diamond$

4.2.2 An Object Calculus

The format of the object calculus definition is similar to the tuple relational calculus definition provided in [Ull82]. A query in the object calculus is of the form $\{o \mid \psi(o)\}$, where o is an object variable denoting some objects in the database and ψ is a formula built from atoms. The result of the query is the set of objects o which satisfy the predicate formed by $\psi(o)$. We introduce a third atom, specific only to calculus expressions, in addition to those defined in the previous section.

Range Atom: $C(o)$ or $C^*(o)$ where C is the name of a class and o is an object variable ranging over the instances of class C. $C(o)$ refers to the objects in the extent of C, i.e., $ext(C)$, whereas $C^*(o)$ refers to the objects in the deep extent of C, i.e., $ext^*(C)$.

Formulas depend on the notion of *free* and *bound* variables. A variable is said to be bound in a formula if it has been previously introduced using a quantifier such as $\exists$ or $\forall$. If the variable has not been introduced using a quantifier it is free in the formula. Formulas are defined as follows:

1. Every atom is a formula. All object variables in the atom are free in the formula.
2. If ψ_1 and ψ_2 are formulas, then $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$ and $\neg\psi_1$ are formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$ and $\neg\psi_1$ as they are free or bound in ψ_1 or ψ_2 depending on where they occur.
3. If ψ is a formula, then $(\exists o)(\psi)$ is a formula. Free occurrences of o in ψ are bound to $(\exists o)$ in $(\exists o)(\psi)$.
4. If ψ is a formula, then $(\forall o)(\psi)$ is a formula. Free occurrences of o in ψ are bound to $(\forall o)$ in $(\forall o)(\psi)$.
5. Formulas may be enclosed in parenthesis. In the absence of parenthesis, the decreasing order of precedence is $\in$, $=$, $=_{\{\}}$, $==$, $\exists$, $\forall$, $\neg$, $\wedge$ and $\vee$, in that order.

A query is an *object calculus expression* of the form $\{o \mid \psi(o)\}$ where o is the only free variable in ψ.

Example 4.2 Using the database of Figure 8, the following sample queries can be formulated as object calculus expressions.

1. Author of the document titled "Principles of Distributed Databases":

$$\{\ o \ \mid\ \exists p(Doc(p) \wedge o ==<p>.author \wedge \text{“}Principles\ of\ Distributed\ Databases\text{”} =<p>.title)\ \}$$

2. Nodes belonging to the document titled "Principles of Distributed Databases":

$$\{\ o \ \mid\ \exists p(Doc(p) \wedge \text{“}Principles\ of\ Distributed\ Databases\text{”} =<p>.title \wedge \exists q(Link^*(q) \wedge p ==<q>.part\_of \wedge (o ==<q>.from \vee o ==<q>.to)))\ \}$$

3. Documents coauthored by a person's father:

$$\{\ o \ \mid\ \exists p(Doc(o) \wedge p ==<o>.author \wedge \exists q(q \in <o>.coauthors \wedge q ==<p>.father))\ \}$$

4. Authors who only write things that include boating among their topics:

$$\{\ o \ \mid\ \forall p(Doc(p)\ \wedge\ o == <p>.author\ \wedge\ \text{"Boating"} \in <p>.keywords)\ \}\quad \diamond$$

At this point it is appropriate to comment on the choice of atoms for the object calculus as some of them seem quite restrictive as compared to the tuple relational calculus. For example, the tuple relational calculus allows the operator θ to be one of $=, <, \leq, >$ or $\geq$ whereas the object calculus restricts θ to $=, ==, =_{\{\}}$ or $\in$. The object calculus allows a value based equality comparison of atomic objects only, not of complex objects.

Some researchers have proposed *shallow* and *deep* equality operators which can be applied to objects of any class [GR83, KC86, LRV88]. Two objects are said to be shallow-equal if their values are identical. Two objects are said to be deep-equal if (1) they are atomic objects and their values are equal, or (2) they are set objects and their elements are pairwise deep-equal, or (3) they are tuple objects and the values they take on the same attributes are deep-equal.

The object calculus defined here avoids these operators for two reasons. First, a value based comparison of complex objects, such as $o_i = o_j$, where o_i and o_j are complex objects, violates the principle of abstract data types whose instances are solely defined by their behavior. In order to completely support encapsulation, one can not allow query expressions whose results are dependent on equivalence of structure as opposed to equivalence of behavior. Second, the model should allow various objects of the same class to be implemented differently to take advantage of their environment. For example, different representations may be used when objects are in main memory versus when they are stored on secondary storage. Furthermore, if distribution and heterogeneity are considered, then objects may be represented differently on different machines. Therefore, the notion of an equivalence test which depends on representation is inappropriate.

A similar argument can be made for prohibiting the use of comparison operators other than $=$ on atomic objects. User knowledge of the values in a domain does not necessarily imply knowledge about their ordering. As an example, consider the case of a Ceasar cipher where all letters are shifted by n characters. With $n = 5$, the encoded form of 'hello world' would be 'czggj rjmgy'. A database might contain the class *CipherAlphabet* whose value domain is the letters of the alphabet and whose total ordering is $< v, \ldots, z, a, \ldots, u >$. Obviously the $<$ relation on members of *CipherAlphabet* is not the same as the $<$ relation on the standard alphabet even though the value domains are identical. For this reason, all value comparison operations other than $=$ must be implemented by a method in a class in accordance with the total ordering the class defines.

4.2.3 An Object Algebra

Operands and results in the object algebra are sets of objects. Thus the algebra maintains the closure property where the result of a query can be used as the input to another. Some of the operators accept more than two operands. Let Θ be an operator in the algebra. The notation $P\ \Theta\ \langle Q_1 \ldots Q_k \rangle$ will be used for algebra expressions where P and Q_i denote sets of objects which are arguments to the operator Θ. In the case where $k = 1$ we will use $P\ \Theta\ Q$ and where $k = 0$ we will use $P\ \Theta\ \langle\ \rangle$ without loss of generality.

Some of the algebra operators are qualified by a predicate. Such operators will be written $P\ \Theta_F\ \langle Q_1 \ldots Q_k \rangle$ where F is a formula consisting of one or more atoms connected by $\wedge$, $\vee$, or $\neg$ using parenthesis as required. Atoms reference lower case, single letter variables which range over objects in the input set named with the corresponding upper case letter. For example, the object variables p, q_1 and q_2 in the predicate of $P\ \Theta_{F(p,q_1,q_2)}\ \langle Q_1, Q_2 \rangle$ range over the sets of objects denoted by P, Q_1 and Q_2 respectively.

The algebra defines five operators.

Union (denoted $P \cup Q$): The union is the set of objects which are in P or Q or both. An equivalent expression for union is $\{ o \mid P(o) \vee Q(o) \}$.

Difference (denoted $P - Q$): The difference is the set of objects which are in P and not in Q. An equivalent expression for difference is $\{ o \mid P(o) \wedge \neg Q(o) \}$. The intersection operator, $P \cap Q$, can be derived by $P - (P - Q)$.

Select (denoted $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$): Select returns the objects denoted by p for each vector $< p, q_1, \ldots, q_k > \in P \times Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for select is $\{ p \mid P(p) \wedge Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge F(p, q_1, \ldots, q_k) \}$.

Multiple operands permit explicit joins as described in [Kim89]. An explicit join is a join between arbitrary classes which support (a sequence of) method applications resulting in comparable objects.

Example 4.3 Find all documents about cars by persons over 50 years of age. Let d range over *Doc* and p range over *Person*, then

$$Doc\ \sigma \left[\begin{array}{c} \text{“}car\text{”} \in <d>.keywords\ \wedge \\ p == <d>.author\ \wedge \\ \text{“50”} = x\ \wedge\ \text{“}True\text{”} = <p,x>.age.greater \end{array} \right] \langle Person \rangle\ \diamond$$

The result of this expression is a set of *Document* objects, not sets of $< Document, Person >$ objects. This is due to the 'object preserving' nature of the algebra which does not support creation of new objects. In this sense then, the select is most like the traditional semi-join operator. As a result, the selection $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$ always returns a subset of P.

Generate (denoted $Q_1\ \gamma_F^t\ \langle Q_2 \ldots Q_k \rangle$): F is a predicate with the condition that it must contain one or more generating atoms for the target variable t and t does not range over any of the argument sets. The operation returns the objects denoted by t in F for each vector $< q_1, \ldots, q_k > \in Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for generate is $\{ t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(Q_k) \wedge F(t, q_1, \ldots, q_k)) \}$.

Two common uses of the generate operator are to collect results of method applications or to iterate over the content of set valued objects.

Example 4.4 Return all co-authors of the document 'My Cat is Object-Oriented' [Kin89]. Let t be the target variable and d range over *Doc*, then

$$Doc\ \gamma^t \left[\begin{array}{c} \text{“}MyCat...\text{”} = <d>.title\ \wedge \\ t \in <d>.co\_authors \end{array} \right] \langle\ \rangle\ \diamond$$

Map (denoted $Q_1 \mapsto_{mlist} \langle Q_2 \ldots Q_k \rangle$): Let *mlist* be a list of method names of the form $m_1 \cdots m_m$. Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in $\langle Q_2 \ldots Q_k \rangle$ as parameters to the methods in *mlist*. This returns the set of objects resulting from each sequence application. If no method in *mlist* requires any parameters, then $\langle Q_2 \ldots Q_k \rangle$ is the empty sequence $\langle\ \rangle$. Map is a special case of the generate operator whose equivalent is $\{ t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge t == < q_1, \ldots, q_k > .mlist \}$. This form of the generate operation warrants its own definition as it occurs frequently and supports several useful optimizations. Map is similar to the **image** operator of [SZ90], except that it is not restricted to unary methods.

4.3 Calculus-Algebra Translation

An algorithm for translating a restricted set of object calculus expressions to their object algebra counterparts is fully described in [SÖ90b]. The algorithm is applicable to *resricted queries*, i.e., those with no occurrences of the universal quantifier $\forall$. This family of queries is similar in power to the select-project-join class of queries in the relational model.

The translation algorithm first rewrites the calculus expression to its prenex disjunctive normal form. Next, the atoms in each conjunct are placed in a candidate list and ranked by whether they restrict the query's target variable, generate the target variable or represent a range atom. The conjunct is then reconstructed as a nested expression by choosing atoms from the candidate list and combining them with atoms referencing common variables and empty expressions representing variables which are unbound in the partially completed conjunct. The process is recursively repeated for each empty expression until the candidate list is exhausted. Each nested subexpression is then mapped to its corresponding algebra expression using a simple pattern matching template.

Several legal orderings of the candidate list may be possible with the result that a family of equivalent algebra expressions is to be obtained. The translation algorithm does not insure optimality of the resulting algebra expressions which still require type checking and logical rewriting. These topics are covered in the following sections.

5 Typechecking of Algebra Expressions

Database query languages have traditionally had only minimal type checking requirements. In the relational model, for example, type checking insures that relation schemes are compatible and that only appropriate comparison operations are performed on tuple fields. The limited number of primitive domains supported by the model (e.g., integer, string, boolean) makes this a straightforward task. OODB query languages introduce complexity into this process as query results may be non-homogeneous sets of objects, i.e., all objects in the query result are not the same type.

Since in closed algebras the result of one query is used as the input to another, there is a need to insure that methods referenced in the predicate of the second query are defined on all objects in the result of the first. Some algebras impose type restrictions such as *union compatibility* [SZ90, Zdo88] on the algebra operators to insure the type consistency of the result. Union compatibility states that members of the sets being operated on must be instances of types which are in a subtype relationship with one another. The type of the result is considered to be the most general supertype of the types involved in the operation.

Another problem, termed *impedance mismatch* [MSOP86], occurs when an application programming language must interface with a database query language. The two languages often have (partially) incompatible data types, e.g., union types in C and relations in SQL. A common requirement, independent of any particular language, is that a program variable be iteratively bound to each element in the set of objects returned by a query, e.g., portals [SR86] and cursors [Ast76]. It should be possible to insure that this binding is type consistent in order to detect improper use of data. This problem becomes more complex when the query results are not homogeneous.

The fundamental question is the following: If the result of a query is a set of objects which may not be homogeneous, what can be said about the types that each member of the query result supports? This is an important question because the intermediate results of a query can be an input to subsequent operators. Therefore, it becomes essential to determine what the minimally common behavior is of the set of objects in the intermediate result to identify which methods can safely be applied to them.

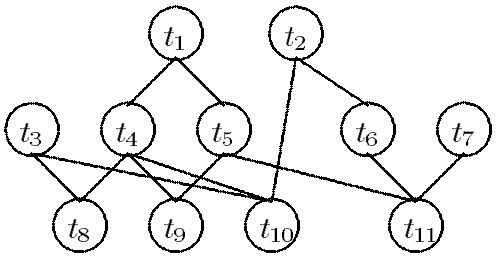

Figure 9: A type lattice fragment.

Example 5.1 Consider the fragment of a type lattice in Figure 9 where types are labeled t_i. Assume we wish to take the union of the instances of types t_8 and t_9. The issue is to determine the behavior of the objects in $ext(t_8) \cup ext(t_9)$. The following can be said about the behavior of these object:

1. Some objects have the behavior of t_3 (immediate supertype of t_8),
2. Some objects have the behavior of t_5 (immediate supertype of t_9),
3. All objects have the behavior of t_4 (immediate supertype of both t_8 and t_9).

Therefore, it is safe to apply the methods defined on t_4 to the objects in $ext(t_8) \cup ext(t_9)$. $\diamond$

The above example demonstrates the need to build a type system to determine the behavior of intermediate query results. The example is simple, but more complicated cases do come up.

5.1 Type Checking Issues

Type checking algebra versus calculus expressions. The type checking system can operate either on calculus expressions or on algebra expressions. The advantage of typechecking calculus expressions is that inconsistencies can be detected early without the query processor doing a significant amount of work in translating a query from its calculus to algebra format. Furthermore, type checking can be integrated with integrity enforcement, which is usually performed up front. However, this requires type checking rules to be defined for general calculus expressions which is not trivial. Typechecking algebra expressions is simpler because it can be reduced to defining type rules for each algebra operator.

It is probably advantageous to apply a combination of calculus versus algebra type checking in a practical query system. It should be possible to define simple type checking rules for the calculus expression to eliminate those which are incorrect. More sophisticated type checking of intermediate query results may be performed on algebraic expressions.

Static versus dynamic type checking. Static type checking applies the type consistency rules at compile time. It has the advantage of identifying errors early and without the potentially harmful results which could occur at run time. However, it also hampers dynamic binding of objects, which is a commonly stated advantage of object-oriented languages. The discussion of static versus dynamic typing in object-oriented languages is an on-going one and query processors will probably be required to accommodate both by doing as much static type checking as possible while providing the means for dynamic binding of variables to objects.

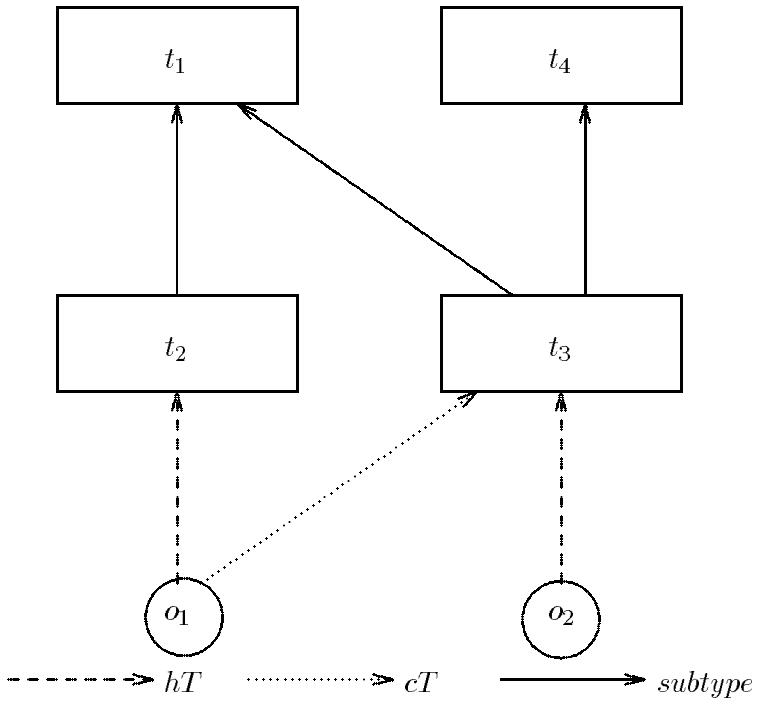

Figure 10: Role of conformance

Basis for type checking. Example 5.1 demonstrates a case where the determination of the behavior of objects in a collection is made on the basis of the type semilattice (i.e., the *subtype* relationship). This is probably the most common approach. However, it is possible to perform type checking by also making use of the *conforms-to* relationship. In this case it may be possible to more "tightly" determine the behavior of objects.

Consider, for example, the case depicted in Figure 10 where the following hold:

1. o_1 *hT* t_2
2. o_2 *hT* t_3
3. o_1 *cT* t_3
4. t_2 *subtype* t_1
5. t_3 *subtype* t_1 and t_3 *subtype* t_4

In this case, if we only depend on the subtype relationship, the common behavior of objects $\{o_1, o_2\}$ is defined by type t_1. However, if the conformance relationship is also used, it can be determined that the common behavior of $\{o_1, o_2\}$ is defined by type t_3. This is preferable since t_3 specializes t_1 and, therefore, has behavior which is more specific than t_1.

We should note that most systems that implement some sort of type checking (e.g. Emerald [BHJ+87]) define conformance relationships between objects and types. However, their definition of conformance is not as general as the one discussed here and is restricted by the subtype/supertype relationship.

5.2 Type checking in the Example Data Model

The type system that we have designed defines a set of type checking rules based on *type conformance* and a type checking methodology based on static type checking of algebra operators. Type conformance in our case is restricted by the subtype/supertype relationship as we discuss below. The typing rules are defined for each algebra operator and for various types of query predicates that are allowed in the model.

We define *conformance* as a set of types. A set of objects O has conformance $\{t_1, \ldots, t_n\}$, denoted by $O{:}\{t_1, \ldots, t_n\}$, when each object $o \in O$ conforms to every type $t_i \in \{t_1, \ldots, t_n\}$.

Example 5.2 Again consider the example in Figure 9. We can now define the behavior of objects in $(ext(t_8) \cup ext(t_9))$ in terms of the conformance relationship:

1. Some objects conform to t_3 (immediate supertype of t_8),
2. Some objects conform to t_5 (immediate supertype of t_9),
3. All objects conform to t_4 (immediate supertype of both t_8 and t_9).

Intuitively then, we may say that the type of $(ext(t_8) \cup ext(t_9))$ is t_4 since this is the only type that all objects in the union conform to. This case is somewhat trivial as all objects in the query result conform to just one class. Referring again to Figure 9, assume we wish to take the union of the instances of types t_{10} and t_{11}. In this case the following can be said about the objects in $(ext(t_{10}) \cup ext(t_{11}))$.

1. Some objects conform to $\{t_3, t_4, t_2\}$ (immediate supertypes of t_{10}),
2. Some objects conform to $\{t_5, t_6, t_7\}$ (immediate supertypes of t_{11}),
3. All objects conform to $\{t_1, t_2\}$ (not necessarily immediate supertypes).

The last statement holds because an object conforms to the type it is an instance of, and via inheritance, any of its supertypes. $\diamond$

We also define a *conformance inclusion relationship* on two sets of types C_1 and C_2 as $C_1 \sqsubseteq C_2$ iff $\forall t_i \in C_2, \exists t_j \in C_1 \mid t_j \preceq t_i$. In other words, $C_1 \sqsubseteq C_2$, if for every type in C_2 there is a conforming type in C_1. Note that C_1 may contain types which do not conform to any type in C_2 under this definition.

The notion of finding the set of types to which all members of a second set of types conforms to is central to determining the type consistency of operations on sets of objects. However, we do not always want to know all the types which are conformed to as this set would contain redundant information. In Example 5.2 the conformance of $(ext(t_{10}) \cup ext(t_{11}))$ was determined to be $\{t_1, t_2\}$. Including parents of t_1 and t_2 in the conformance would add no new type information since t_1 and t_2 define at least, if not more than, the behavior of their parents, i.e., t_1 and t_2 are specializations of their parent types. Similarly, placing more general types in the conformance, for example parents of t_1 and t_2 but not t_1 or t_2 themselves, introduces a loss of type information.

Loss of type information is undesirable when type checking a query. Consider again the type lattice fragment of Figure 9. Assume all objects in a query result conform to both t_{10} and t_{11} but the conformance was nonetheless specified as $\{t_1, t_2\}$. This would correspond to the case where types more general than necessary are placed into the conformance. It is possible that the query in question was just a subquery and that further operations are to be performed on its result. Some of the object algebra operators are qualified by predicates. One form of predicate involves applying

a method to each member in the query set. If the method referenced in the query is defined on t_{11} but not on t_2, the query will fail during type checking when in fact each member of the query set does support that method. Thus we have the requirement that the conformance of a set of objects used in type checking include only the most specific types which satisfy the conformance definition.

The conformance of a set of objects O, $O{:}\{t_1, \ldots, t_n\}$, is defined to be the *most specific conformance* when there does not exist a subtype $s \preceq t_i$ such that all elements of O conform to s. The function $MSC(t_1, \ldots, t_n)$ is defined to return the most specific conformance of the types $t_1, \ldots, t_n$.

Example 5.3 Referring to Figure 9:

$$\begin{aligned} MSC(t_{10}) &\equiv \{t_{10}\} \\ MSC(t_{10}, t_{11}) &\equiv \{t_1, t_2\} \\ MSC(t_{10}, t_6) &\equiv \{t_2\} \ \diamond \end{aligned}$$

The need will arise during type checking to determine the inverse MSC relationship. Letting s and t refer to subtypes and types respectively, the function MSC^{-1} is defined as

$$MSC^{-1}(t_1, \ldots, t_n) \equiv \{\ s_1, \ldots, s_k \mid MSC(s_1, \ldots, s_k) = \{t_1, \ldots, t_n\}\ \}$$

In other words, the inverse function MSC^{-1} returns the most general set of subtypes all of whom conform to $t_1, \ldots, t_n$.

Example 5.4 Referring to Figure 9:

$$\begin{aligned} MSC^{-1}(t_1) &\equiv \{t_1\} \\ MSC^{-1}(t_1, t_2) &\equiv \{t_{10}, t_{11}\} \\ MSC^{-1}(t_5, t_7) &\equiv \{t_{11}\} \ \diamond \end{aligned}$$

The full set of type conformance rules that form the foundation of this type checking methodology is given in [SÖ90d]. The rules determine the conformance of an expression from the conformance(s) of its subexpressions. A type checking algorithm is correct if it computes types that are derivable by these rules. An expression is considered type inconsistent if the rules can not be used to derive a type (conformance) for all variables in the expression.

6 Algebraic Optimization

The next step in the query processing methodology depicted in Figure 1 is the "optimization" of the type consistent algebra expressions. This is accomplished by means of equivalence-preserving transformation rules. The fundamental design issue is to make sure that a complete set of transformation rules have been captured for the object algebra and that they are efficiently used by a query processor.

The overall goal of algebraic "optimization" is to reduce the cost of evaluating a query by replacing its algebraic expression with one which may have better performance characteristics. Haas et. al. [HFLP89] make the distinction between two rule based query transformation techniques, "query rewrite" and "plan optimization". Query rewrite is a high level process where general purpose heuristics drive the application of transformation rules. Plan optimization is a lower level process which transforms a query into the most cost effective access plan based on a specific cost model and knowledge of access paths and database statistics. In this section we discuss rules that are intended for use during query rewrite. In the next section we address issues related to plan optimization.

We demonstrate the idea of algebraic optimization using rewriting rules by discussing the full set of rules that have been specified for the object algebra that was presented before. The proofs of these rules are presented in [Str91]. The rules are grouped into *algebraic* and *semantic* ones. Algebraic rules create equivalent expressions based upon pattern matching and textual substitution. Semantic rules are similar, but they are additionally dependent on the semantics of the database schema as defined by the class definitions and inheritance lattice.

Rules will be written as $E_1 \Leftrightarrow E_2$ which specifies that expression E_1 is equivalent to expression E_2. As in [Fre87] we also use restricted rules of the form $E_1 \overset{c}{\Leftrightarrow} E_2$. Restricted rules are applicable only when the condition c is true. Conditions are a conjunction of functions which determine properties of argument sets, predicates and variables used in a rule. We define the function $\text{ref}(F,(v_1,\ldots,v_n))$ to be true when $v_1,\ldots,v_n$ are the *only* variables referenced in the predicate F. The function $\text{gen}(F,v)$ tests whether the predicate F contains a generating atom for the variable v. Similarly, $\text{res}(F,v)$ is true when predicate F restricts values of v. For example, $\text{gen}(F,t)$ is true when $F \equiv (t \in <q_1,q_2>.mlist)$ and false when $F \equiv (q_2 =_{\{\}} <t,q_1>.mlist)$.

We use the following notation for select/generate sets:

$$\{\ p \mid \exists q_1 \cdots \exists q_k\ F(p,q_1,\ldots,q_k)\ \} \equiv \begin{cases} P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle & \text{iff res}(F,p) \\ Q_1\ \gamma_F^p\ \langle Q_2 \ldots Q_k \rangle & \text{iff gen}(F,p) \end{cases}$$

In other words, the set definition for select and generate operations can only be distinguished by the properties of predicate F. If F is defined as restricting values of p, then the operation is a select. If F is defined as generating values for p, then the operation is a generate. □

An arbitrary expression in a list of expressions is referenced using the notation $(E_1 \ldots E_i \ldots E_n)$ where '...' denotes zero or more occurrences of some E_i. For example, the rule

$$P\ \sigma_F\ \langle \ldots Q_x \ldots Q_y \ldots \rangle \quad \Leftrightarrow \quad P\ \sigma_F\ \langle \ldots Q_y \ldots Q_x \ldots \rangle \tag{1}$$

indicates that the result of a select operation is independent of the ordering of the arguments between '⟨' and '⟩'. The set being restricted must appear before the select operator σ_F, thus there is no rule to change the position of P. This is not the case for generate operations as the target variable does not correspond to one of the input sets. The next two rules state that the outcome of a generate is independent of the operand ordering.

$$Q_x\ \gamma_F^t\ \langle \ldots Q_y \ldots \rangle \quad \Leftrightarrow \quad Q_y\ \gamma_F^t\ \langle \ldots Q_x \ldots \rangle \tag{2}$$

$$Q_x\ \gamma_F^t\ \langle \ldots Q_y \ldots Q_z \ldots \rangle \quad \Leftrightarrow \quad Q_x\ \gamma_F^t\ \langle \ldots Q_z \ldots Q_y \ldots \rangle \tag{3}$$

We introduced the map operator as a special case of generate. This can be captured by the conditional rule:

$$Q_1\ \gamma_F^t\ \langle Q_2 \ldots Q_k \rangle \quad \overset{c}{\Leftrightarrow} \quad Q_1\ \mapsto_{mlist}\ \langle Q_2 \ldots Q_k \rangle \tag{4}$$

where condition c insures that $F \equiv (t == <q_1,\ldots,q_k>.mlist)$.

We introduce the abbreviations $Qset$, $Rset$ and $Sset$ to replace $Q_1 \ldots Q_k$, $R_1 \ldots R_l$ and $S_1 \ldots S_m$ respectively. For example:

$$P\ \sigma_F\ \langle Qset, Rset, Sset \rangle \quad \overset{c}{\Leftrightarrow} \quad P\ \sigma_F\ \langle Q_1 \ldots Q_k, R_1 \ldots R_l, S_1 \ldots S_m \rangle \tag{5}$$

where condition c is $\text{ref}(F,(p,q_1,\ldots,q_k,r_1,\ldots,r_l,s_1,\ldots,s_m))$. As before, a lower case letter represents an object variable which ranges over the set denoted by the corresponding upper case letter, (i.e., $q_i \in Q_i$, $r_i \in R_i$ and $s_i \in S_i$).

6.1 Object Algebra Identities

Since the operands and the results of algebra operators are sets of objects, where set membership is determined by object identity, the typical set-theoretic identities of typical set-theoretic algebra operators (union, intersection and difference) apply. Among others, these identities include the associativity of union and intersection, and the distribution of union, intersection and difference operators over each other.

Additionally, the following rules specify other identities in the object algebra, i.e., there are no conditions associated with them.

$$(P\ \sigma_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_{F_2}\ \langle Rset\rangle)\ \sigma_{F_1}\ \langle Qset\rangle \tag{6}$$

$$(P - Q)\ \sigma_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset\rangle) - Q \tag{7}$$

$$(P \cup Q)\ \sigma_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset\rangle) \cup (Q\ \sigma_F\ \langle Rset\rangle) \tag{8}$$

$$(P \cap Q)\ \sigma_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset\rangle) \cap (Q\ \sigma_F\ \langle Rset\rangle) \tag{9}$$

$$(P \cup Q)\ \mapsto_{mlist}\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \mapsto_{mlist}\ \langle Rset\rangle) \cup (Q\ \mapsto_{mlist}\ \langle Rset\rangle) \tag{10}$$

$$(P \cup Q)\ \gamma^t_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \gamma^t_F\ \langle Rset\rangle) \cup (Q\ \gamma^t_F\ \langle Rset\rangle) \tag{11}$$

Rule 6 captures commutativity of select. Rules 7-9 show that difference, union and intersection commute with select. Rule 10 specifies that union commutes with map and the last rule indicates that union distributes over generate. These rules are not as general as they appear since the leading argument of a select or map operation can not be swapped with one of the trailing arguments. The following rules capture the commutativity of union in a trailing argument.

$$P\ \sigma_F\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k\rangle$$
$$\Leftrightarrow\ (P\ \sigma_F\ \langle Q_1 \ldots Q_x \ldots Q_k\rangle)\ \cup\ (P\ \sigma_F\ \langle Q_1 \ldots Q_y \ldots Q_k\rangle) \tag{12}$$

$$P\ \gamma^t_F\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k\rangle$$
$$\Leftrightarrow\ (P\ \gamma^t_F\ \langle Q_1 \ldots Q_x \ldots Q_k\rangle)\ \cup\ (P\ \gamma^t_F\ \langle Q_1 \ldots Q_y \ldots Q_k\rangle) \tag{13}$$

$$P\ \mapsto_{mlist}\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k\rangle$$
$$\Leftrightarrow\ (P\ \mapsto_{mlist}\ \langle Q_1 \ldots Q_x \ldots Q_k\rangle) \cup (P\ \mapsto_{mlist}\ \langle Q_1 \ldots Q_y \ldots Q_k\rangle) \tag{14}$$

Example 6.1 Consider the query "Return the root nodes of all documents which are either about cats or about dogs". Let

d	range over the class *Document*
F_1	be the atom ("*cats*" $\in$<*d*>.*keyWords*)
F_2	be the atom ("*dogs*" $\in$<*d*>.*keyWords*)
n	range over *Node* objects

Then we can use the following object algebra expression to implement the query

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle))\ \mapsto_{rootNode}\ \langle\ \rangle$$

and apply rule 10 to get

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle)\ \mapsto_{rootNode}\ \langle\ \rangle) \cup ((Doc\ \sigma_{F_2}\ \langle\ \rangle)\ \mapsto_{rootNode}\ \langle\ \rangle)$$

The transformation is shown graphically on the right hand side of Figure 11. $\diamond$

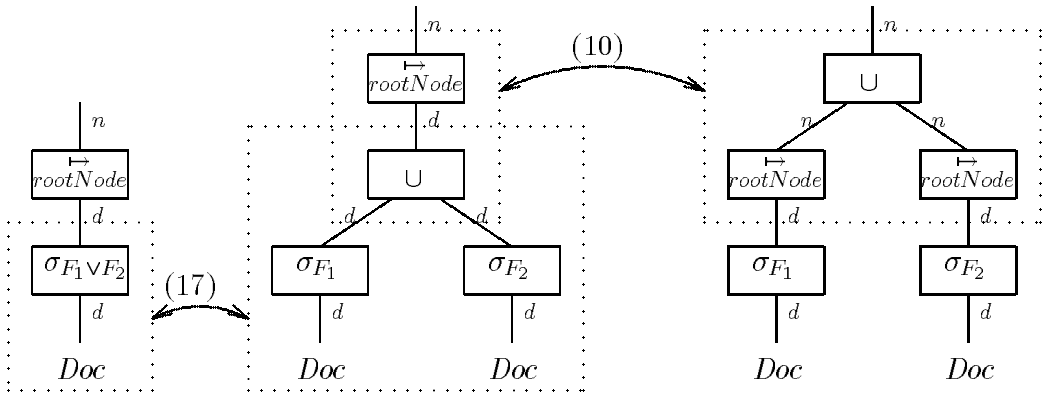

Figure 11: Transformations of examples 6.1 and 6.2.

6.2 Select Transformation Rules

Select supports several transformations, some of them conditional:

$$(P\ \sigma_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_{F_1}\ \langle Qset\rangle)\ \cap\ (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{15}$$

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, Rset\rangle \quad \stackrel{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset\rangle) \cap (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{16}$$

$$P\ \sigma_{(F_1 \vee F_2)}\ \langle Qset, Rset\rangle \quad \stackrel{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset\rangle) \cup (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{17}$$

where:

$$c:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k))\ \wedge\ \mathrm{res}(F_1, p)\ \wedge\ \mathrm{ref}(F_2, (p, r_1 \ldots r_l))\ \wedge\ \mathrm{res}(F_2, p)$$

Rule 15 is an identity which utilizes the fact that selection merely restricts its input and returns a subset of its first argument. The first selection, $P\ \sigma_{F_1}\ \langle Qset\rangle$, returns a subset of P (call it P'). The second selection can then be reduced to $P'\ \sigma_{F_2}\ \langle Rset\rangle$ which is merely a smaller subset of P. The same final subset of P can be obtained by applying predicates F_1 and F_2 separately and taking the intersection of the results.

Rules 16 and 17 recognize that subformulas F_1 and F_2 each reference only a subset of the arguments. Operand sizes are minimized by breaking F_1 and F_2 into separate select operations and intersecting $(F_1 \wedge F_2)$ or taking the union $(F_1 \vee F_2)$ of the results.

Example 6.2 The union subquery $((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle))$ of Example 6.1 matches the right hand side of rule 17. Substituting Doc for P and $\langle\ \rangle$ for $Qset$ and $Rset$ we can apply rule 17 right to left resulting in

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle)) \quad \Leftrightarrow \quad Doc\ \sigma_{F_1 \vee F_2}\ \langle\ \rangle$$

This transformation is shown graphically on the left hand side of Figure 11. $\diamond$

Noting the similarities in rules 15 and 16 allows us to derive a new rule for conjunctive predicates:

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, Rset\rangle \quad \stackrel{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \tag{18}$$

where:

$$c:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k))\ \wedge\ \mathrm{res}(F_1, p)\ \wedge\ \mathrm{ref}(F_2, (p, r_1 \ldots r_l))\ \wedge\ \mathrm{res}(F_2, p)$$

A slightly different version of the conjunctive predicate rule for select is:

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset\rangle \overset{c_1}{\Leftrightarrow} P\ \sigma_{F_1}\ \langle Qset, (R\ \sigma_{F_2}\ \langle Sset\rangle)\rangle \quad (19)$$

$$\overset{c_2}{\Leftrightarrow} P\ \sigma_{F_1}\ \langle Qset, (R\ \gamma^{t}_{F_2}\ \langle Sset\rangle)\rangle \quad (20)$$

where:

$$c_1:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k, r))\ \wedge\ \mathrm{ref}(F_2, (r, s_1 \ldots s_m))\ \wedge\ \mathrm{res}(F_2, r)$$
$$c_2:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k, t))\ \wedge\ \mathrm{ref}(F_2, (t, r, s_1 \ldots s_m))\ \wedge\ \mathrm{gen}(F_2, t)$$

Here, F_1 and F_2 have only one variable in common, and it is not the variable being restricted by the select. In the first rule, the common variable is r which is restricted by F_2. Since F_1 does not reference $s_1 \ldots s_m$ and F_2 does not reference $p, q_1 \ldots q_k$, the restriction on R can be pushed down into a separate select operation. The second rule is similar, however we denote by t the common variable shared by F_1 and F_2 to reflect that it is the target variable generated by F_2. In this case F_2 is pushed down into a separate generate operation.

Example 6.3 Consider the query "Find all documents written by the child of a computer scientist and a doctor". Let

d	range over the class *Document*
p_1	range over the class *Person*
p_2	range over the class *Person*
c	range over *Person* objects which are children
a_1	be the atom (c ==<d>.*author*)
a_2	be the atom ("*computers*" ∈<p_1>.*expertise*)
a_3	be the atom ("*medicine*" ∈<p_2>.*expertise*)
a_4	be the atom (c ∈<p_1, p_2>.*children*)

The following object algebra expression can be used to represent the query

$$Doc\ \sigma_{(a_1 \wedge a_2 \wedge a_3 \wedge a_4)}\ \langle Person, Person\rangle$$

This expression satisfies the conditions of rule 20 when we substitute a_1 for F_1, $(a_2 \wedge a_3 \wedge a_4)$ for F_2, *Doc* for P, { } for *Qset*, *Person* for R and *Person* for *Sset*. Applying rule 20 with these substitutions gives

$$Doc\ \sigma_{(a_1 \wedge a_2 \wedge a_3 \wedge a_4)}\ \langle Person, Person\rangle \quad \Leftrightarrow \quad Doc\ \sigma_{a_1}\ \langle (Person\ \gamma^{c}_{(a_1 \wedge a_2 \wedge a_3)}\ \langle Person\rangle)\rangle$$

This transformation is shown graphically on the left hand side of Figure 12. ◇

A special case of rule 20 occurs when the select predicate contains a generating atom for the set being restricted (Figure 13).

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset\rangle \overset{c}{\Leftrightarrow} (p\ \sigma_{F_1}\ \langle Qset\rangle) \cap (R\ \gamma^{p}_{F_2}\ \langle Sset\rangle) \quad (21)$$

where:

$$c:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k))\ \wedge\ \mathrm{res}(F_1, p)\ \wedge\ \mathrm{ref}(F_2, (p, r, s_1 \ldots s_m))\ \wedge\ \mathrm{gen}(F_2, p)$$

The left hand side of this rule indicates that elements of P are restricted by the predicate $(F_1 \wedge F_2)$. However, if F_1 restricts p while F_2 generates p, then we really have two sources of values for p: the argument set P, and the generating atom in F_2. Since the result of the operation must satisfy the restriction of P by F_1 as well as the generation of p by F_2, we can break F_1 and F_2 into separate operations and take the intersection of the result.

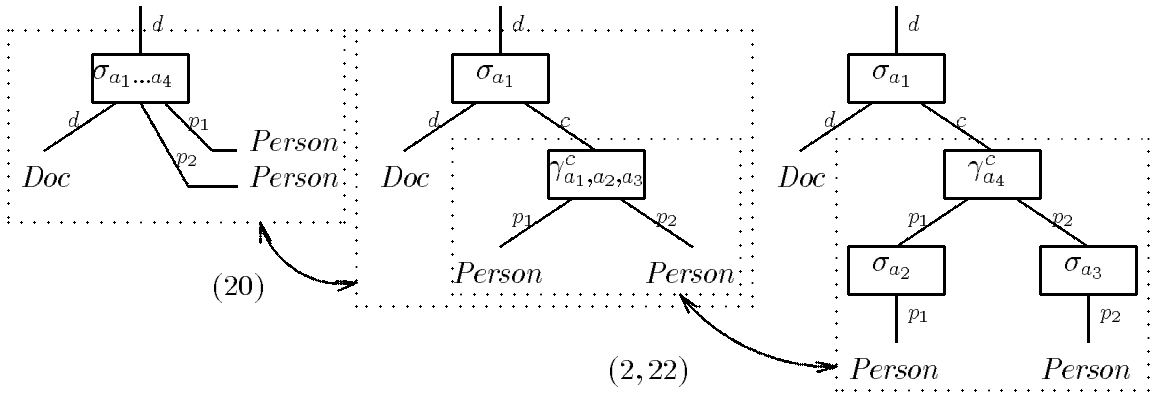

Figure 12: Transformations of examples 6.3 and 6.4.

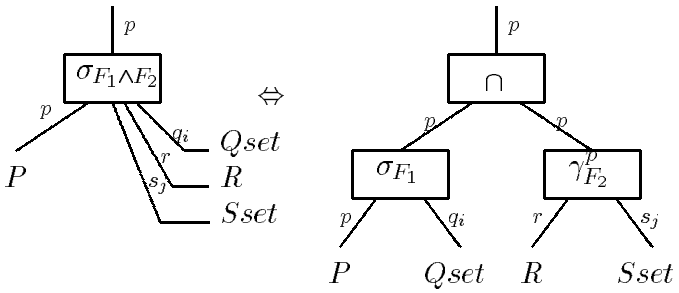

Figure 13: Graphical representation of rule 20.

6.3 Generate Transformation Rules

Conjunctive predicates give rise to several transformation rules for generate operations:

$$P\ \gamma^t_{(F_1 \wedge F_2)}\ \langle Qset, Rset\rangle \overset{c_1}{\Leftrightarrow} (P\ \sigma_{F_1}\ \langle Qset\rangle)\ \gamma^t_{F_2}\ \langle Rset\rangle \quad (22)$$

$$\overset{c_2}{\Leftrightarrow} (P\ \gamma^u_{F_1}\ \langle Qset\rangle)\ \gamma^t_{F_2}\ \langle Rset\rangle \quad (23)$$

$$\overset{c_3}{\Leftrightarrow} (P\ \gamma^t_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \quad (24)$$

where:

$$c_1: \text{ref}(F_1, (p, q_1 \ldots q_k)) \wedge \neg\text{gen}(F_1, p) \wedge \text{ref}(F_2, (p, t, r_1 \ldots r_l)) \wedge \text{gen}(F_2, t)$$
$$c_2: \text{ref}(F_1, (p, u, q_1 \ldots q_k)) \wedge \text{gen}(F_1, u) \wedge \text{ref}(F_2, (u, t, r_1 \ldots r_l)) \wedge \text{gen}(F_2, t)$$
$$c_3: \text{ref}(F_1, (p, q_1 \ldots q_k, t)) \wedge \text{gen}(F_1, t) \wedge \text{ref}(F_2, (r_1 \ldots r_l, t) \wedge \text{res}(F_2, t)$$

The conditions for rule 22 insure that F_1 restricts p, which is the common variable between the two conjuncts of the predicate. F_1 is subsequently broken out into a separate select operation. In rule 23, u is the common variable and is generated by F_1 which can also be broken out into a separate generate operation. In rule 24, t is the common variable between the two conjuncts of the predicate. F_1 generates values for t while F_2 merely restricts t which allows the conjuncts to be broken out into separate generate and select operations.

Example 6.4 Consider the subquery $Person\ \gamma^c_{a_2 \wedge a_3 \wedge a_4}\ \langle Person\rangle$ of Example 6.3 where

$$a_2 \equiv \text{“}computers\text{”} \in <p_1>.expertise$$
$$a_3 \equiv \text{“}medicine\text{”} \in <p_2>.expertise$$
$$a_4 \equiv c \in <p_1, p_2>.children$$

which returns the children which have a doctor and computer scientist as parents. The subquery satisfies the conditions of rule 22 when we substitute a_2 for F_1, $a_3 \wedge a_4$ for F_2, *Person* for P, c for t, { } for *Qset* and *Person* for *Sset* resulting in the transformation

$$Person\ \gamma^{c}_{a_2 \wedge a_3 \wedge a_4}\ \langle Person \rangle \quad \Leftrightarrow \quad (Person\ \sigma_{a_2}\ \langle\ \rangle)\ \gamma^{c}_{a_3 \wedge a_4}\ \langle Person \rangle$$

Ideally we would like to apply this rule again to break out atom a_3 which restricts the variable p_2 ranging over the class *Person*. Noting that the ordering of argument sets does not affect the result of a generate operation, we can apply rule 2 to give

$$(Person\ \sigma_{a_2}\ \langle\ \rangle)\ \gamma^{c}_{a_3 \wedge a_4}\ \langle Person \rangle \quad \Leftrightarrow \quad Person\ \gamma^{c}_{a_3 \wedge a_4}\ \langle (Person\ \sigma_{a_2}\ \langle\ \rangle) \rangle$$

Now we can apply rule 22 again to break out atom a_3 as follows.

$$Person\ \gamma^{c}_{a_3 \wedge a_4}\ \langle (Person\ \sigma_{a_2}\ \langle\ \rangle) \rangle \quad \Leftrightarrow \quad (Person\ \sigma_{a_3}\ \langle\ \rangle)\ \gamma^{c}_{a_4}\ \langle (Person\ \sigma_{a_2}\ \langle\ \rangle) \rangle$$

The result of these steps is shown on the right hand side of Figure 12. $\diamond$

A special case of the generate operation can occur when the predicate generates values for the target variable (which does not range over an argument set) and also for a variable which does range over an argument set.

$$P\ \gamma^{t}_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset \rangle \quad \stackrel{\mathcal{E}}{\Leftrightarrow} \quad ((P\ \gamma^{r}_{F_1}\ \langle Qset \rangle) \cap R)\ \gamma^{t}_{F_2}\ \langle Sset \rangle \tag{25}$$

where:

$$c:\ \mathrm{ref}(F_1, (p, r, q_1 \ldots q_k))\ \wedge\ \mathrm{gen}(F_1, r)\ \wedge\ \mathrm{ref}(F_2, (r, t, s_1 \ldots s_m))\ \wedge\ \mathrm{gen}(F_2, t)$$

The condition states that F_1 generates values for r while F_2 generates values for the target variable t. Similar to rule 21, we now have two sources of values for r; the argument set R and the generating atom in F_1. Since the final values of r must exist in R and be generated by F_1, we can break F_1 out into its own generate operation and intersect the result with R prior to generating values for t.

6.4 Semantic Transformations

Semantic transformation rules take advantage of the semantics of the object-oriented data model. The database schema, as defined by class definitions and the inheritance lattice, captures many relationships which can be used to simplify object algebra expressions. For example, let c_1 and c_2 represent classes, C_i represent the set of objects in the extent of class c_i, $ext(c_i)$, and C_i^* represent the deep extent of c_i, $ext^*(c_i)$. We can show that the expression $C_1 \cap C_2 = \phi$ when $c_1 \neq c_2$ by noting that the data model restricts each object to membership in a single class.

We define two relationships on classes (not class extents) to assist in categorizing the special cases where simplifications are possible. The case where c_1 is a subclass of c_2 is denoted by $c_1 \preceq c_2$. We also use $c_1 \bigtriangledown c_2$ to denote that c_1 and c_2 have subclasses in common, i.e., $ext^*(c_1) \cap ext^*(c_2) \neq \phi$. Conversely, $c_1 \bigtriangledown c_2$ implies that there exists a class c_i which has both c_1 and c_2 as superclasses. These two relationships can be used to derive the special cases of the binary object algebra operators shown in Table 4.

Other semantic rules rely on type consistency to determine their applicability. Consider the following rules (Figure 14).

$$(P\ \sigma_F\ \langle Qset \rangle) \cap R \quad \stackrel{\mathcal{E}}{\Leftrightarrow} \quad (P\ \sigma_F\ \langle Qset \rangle) \cap (R\ \sigma_{F'}\ \langle Qset \rangle) \tag{26}$$

$$\stackrel{\mathcal{E}}{\Leftrightarrow} \quad P \cap (R\ \sigma_{F'}\ \langle Qset \rangle) \tag{27}$$

where:

Table 4: Special cases of the binary operators.

Rule	Condition
$C_1 \cap C_2 = \phi$	$c_1 \neq c_2$
$C_1 - C_2 = C_1$	$c_1 \neq c_2$
$C_1 \cap C_2^* = C_1$	$c_1 \preceq c_2$
$C_1 \cap C_2^* = \phi$	$c_1 \bigtriangledown c_2$
$C_1 \cup C_2^* = C_2^*$	$c_1 \preceq c_2$
$C_1 - C_2^* = C_1$	$c_1 \neq c_2 \wedge c_1 \not\preceq c_2$
$C_1^* - C_2 = C_1^*$	$c_1 \bigtriangledown c_2$
$C_1^* \cap C_2^* = \phi$	$c_1 \bigtriangledown c_2$
$C_1^* \cap C_2^* = \cup C_i$	$c_i \preceq c_1 \wedge c_i \preceq c_2 \wedge c_1 \bigtriangledown c_2$
$C_1^* \cup C_2^* = C_2^*$	$c_1 \preceq c_2$
$C_1^* - C_2^* = C_1^*$	$c_1 \bigtriangledown c_2$

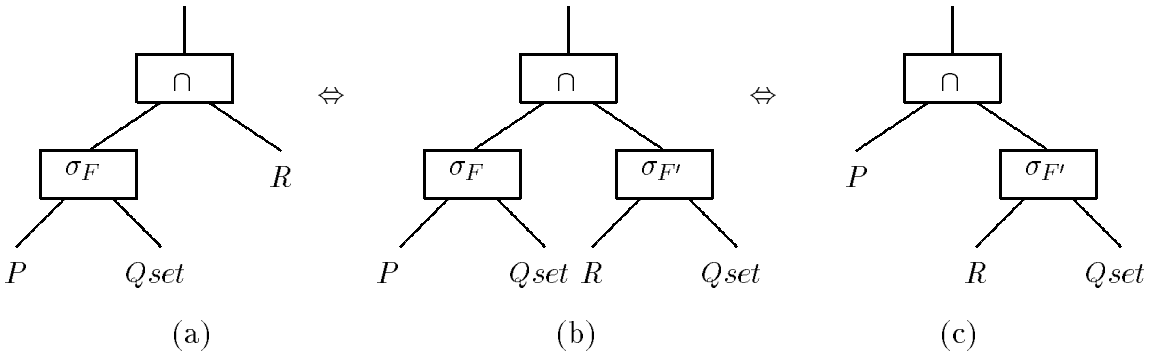

Figure 14: A semantic transformation rule

c: F' is identical to F except each occurrence of p is replaced by r.

By definition, intersection returns only those objects which are present in both input sets. From a pure set theory perspective, any restriction which removes objects from one input set only, automatically excludes those objects from the result of the intersection. In addition, the restriction could be applied equally well to the other input set instead and generate the same result. This is because intersection is not dependent on operand ordering and "doesn't care" which input set is missing the excluded objects. By a similar argument, the restriction could be applied to both input sets without affecting the result.

Referring to Figure 14, the restriction in (a) is given by the select operation $P\ \sigma_F\ \langle Qset \rangle$; i.e., the input set P is being restricted by predicate F. Let us first examine the transformation from (a) to (b).

The expression depicted in (a) is considered a type consistent expression if the methods in predicate F are defined on all types represented by the objects in P. Note that P (R) may be a heterogeneous set of objects if it represents the result of a subquery as opposed to the extent of some class. The transformation from (a) to (b) is valid only if the expression in (b) is a legal. This means the subquery $R\ \sigma_{F'}\ \langle Qset \rangle$ in (b) must be type consistent, i.e., the methods in predicate F' are defined on all types represented by the objects in R. According to condition c on the transformation rule though, the methods in F' are identical to those in F. Thus, the validity of the transformation from (a) to (b), and by similar argument from (c) to (b), is dependent on both the database and the nature of the subqueries which produce P and R.

Once these transformations are shown to be valid, we can conclude that the equivalence of expressions (a) and (c) is valid by the characteristics of intersection discussed previously. If the

restriction can legally be applied to both input sets, then intersection only requires that it be applied to one of them.

7 Execution Plan Generation

In this section we address the issues that arise in the last step of the methodology depicted in Figure 1. This step involves the generation of alternative execution plans for the algebra expression that is obtained after the application of the rewrite rules. This step is commonly called the *access path selection* in relational DBMSs and concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In OODBMSs, the issue is more complicated due to the difference in the abstraction levels of behaviorally defined objects and their storage. Therefore, we separate the issues of *execution plan generation*, which is the mapping of object algebra expressions to some object manager interface expressions, and *access plan selection*, which involves the selection of the optimum execution plan and the efficient implementation of the object manager interface operations. In one sense, this is similar to query processing in distributed databases systems [ÖV91] which involves both global plan generation and local optimization. In this section, we are mainly concerned with execution plan generation.

There are two primary concerns in generating execution plans. The first is to decompose the object algebra operators union ($\cup$), difference ($-$), select (σ_F), map ($\mapsto_{mlist}$) and generate (γ_F^t) (especially those with complex predicates) into a sequence of simpler operations which more accurately reflect the interface provided by a real object-based system. In other words, we are defining a lower level of abstraction than that provided by the data model and object algebra thus far, and treat access plan generation as the mapping of object algebra expressions to the new abstraction interface. This lower level is the object management interface. The second concern is that we still wish to maintain the data abstraction provided by behaviorally defined objects and do not want to make assumptions about how objects are stored and implemented.

7.1 Object Manager Design Issues

In the case of the relational data model [Cod70], there is a close correspondence between algebra operations and the low level primitives of the physical system [SAC+79]. The mapping between relations and files, and tuples and records may have contributed to this strong correspondence. However, there is no analogous, intuitive correspondence between object algebra operators and physical system primitives. Thus any discussion of execution plan generation must first define the low level object manipulation primitives which will be the building blocks of execution plans. We call this low level object manipulation interface the Object Manager (OM) interface.

Object managers have received attention lately in the context of distributed systems [BHJL86, DLA88, MG89, VKC86], programming environments [Dec86, Kae86, VBD89] and databases [CDRS86, CM84, EE87, HZ87, KBC+88]. These object managers are typically oriented towards "one-at-a-time" object execution and differ in terms of their support for data abstraction, concurrency and object distribution.

Encapsulation of objects, which hides their implementation details, and the optimization of queries against these objects pose a challenging design problem which can simply be stated as follows: "At what point in query processing should the query optimizer access information regarding the storage of objects?" We differentiate between two types of object storage information: *representation information*, which specifies the data structures used to represent objects themselves, and *physical storage information* regarding the clustering of objects, indexes defined on them, etc. If object storage is under the control of an object manager, the design question can be posed in terms

of the level of OM interface. Physical optimization of query executions requires storage information, arguing for a high-level OM interface that is accessed early in the optimization process. Many systems that are typically called "complex object systems" choose this approach. Encapsulation, on the other hand, hides storage details and, therefore, argues for a low-level OM interface that is accessed late in the process.

7.1.1 OM Design Principles

Since our data model treats objects as instances of abstract data types, encapsulation is a fairly important consideration. Furthermore, in our work, we are interested in investigating how far we could go with query processing without accessing the physical storage information. Therefore, we have elected to define a fairly low-level OM interface that is accessed late in the optimization process. Furthermore, the OM interface does not reveal any physical organization information. In other words, we are defining a lower level of abstraction than that provided by the data model and object algebra.

Object algebra expressions which are the input to the execution plan generation process have several important characteristics:

1. They can be represented as a graph whose nodes are object algebra operators and whose edges represent streams (sets) of objects. Thus intermediate results do not have any structure. In fact, the intermediate results can be thought of as streams of individual object identifiers.

2. Some algebra operators (σ_F, γ_F^t) are qualified by a predicate. Predicates are formed as a conjunction of atoms, each of which may reference several variables. The variable corresponding to the result of the algebra operation is called the *target variable*.

3. A variable name appearing in multiple atoms of a predicate implies a 'join' of some kind; i.e., objects denoted by the variable must satisfy several conditions concurrently.

The last point, namely implied 'joins' between object variables within a predicate, is the driving factor behind our query execution and execution plan generation strategy. Consider the predicate F for the select operation $P\ \sigma_F\ \langle O, R, S, T\rangle$

$$F \equiv o == (<p, q, r>.m_1) \wedge (q \in t) \wedge (q ==<s>.m_2) \tag{28}$$

where p is the target variable and O, P, R, S, T are inputs to the operation. All values for q are generated by the atoms in the predicate. The result of this select operation can be defined as

$$\{o | F(o, p, q, r, s, t) \text{ is true for } <o, p, r, s, t> \in O \times P \times R \times S \times T\ \} \tag{29}$$

Table 5 identifies which variables are referenced in each atom (numbered left to right) and reflects the dependencies between the variables. It should be clear from the table that an object denoted by

	o	p	q	r	s	t	
$a1$	x	x	x	x			$(o ==<p, q, r>.m_1)$
$a2$			x			x	$(q \in t)$
$a3$			x		x		$(q ==<s>.m_2)$

Table 5: Dependencies between variables in a predicate.

q must satisfy all atoms concurrently. However, if we are to respect the data abstraction afforded by objects, then it is not possible for the query processor to directly evaluate all three atoms

concurrently as required. Instead, it is more likely that we call upon another agent which can perform individual operations on objects that correspond to the individual atoms. This would then require the ability to keep track of the combinations of variables in $O \times P \times R \times S \times T$ which satisfy F. This intuition leads to the following design decisions.

1. The low level operators used to generate an execution plan for an algebra level operator will consume and generate streams (sets) of tuples of object identifiers. We introduce the notation $[a, b, c, \cdots]$ to denote a stream of tuples of object identifiers of the form $\{<a, b, c, \cdots>\}$. For convenience we will call this an *oid-stream* in the remainder of the document. This way relationships among variables and the atoms they satisfy can be maintained over a sequence of operations.

2. The object manager interface performs low level operations comparable to individual atoms in a predicate.

7.1.2 OM Interface Specification

The object manager interface specifies a calling sequence and semantics for performing operations on oid-streams. Four operation types are defined:

1	$\mathbf{OM}_{\cup}([i_1], [i_2], [o])$	– stream union
2	$\mathbf{OM}_{\mathit{diff}}([i_1], [i_2], [o])$	– stream difference
3	$\mathbf{OM}_{\mathit{eval}}([i_1], \ldots, [i_n], [o], \mathit{meth}, \mathit{pred})$	– atom evaluation
4	$\mathbf{OM}_{\bowtie}([i_1], \ldots, [i_n], [o])$	– stream reduction

where $[i_n]$ and $[o]$ denote input and output oid-streams respectively. The semantics of the OM calls are described next.

(1) Stream Union: This operator generates the union of the two input oid-streams. Streams $[i_1]$ and $[i_2]$ must reference the same variable names though not necessarily in the same order. The operation is analogous to the relational union operator. The output oid-stream contains those tuples which are present in $[i_1]$ or $[i_2]$ projected onto the variables identified by the output specifier $[o]$.

(2) Stream Difference: This operator generates the difference of the two input oid-streams. Streams $[i_1]$ and $[i_2]$ must reference the same variable names though not necessarily in the same order. The operation is analogous to the relational difference operator. The output oid-stream contains those tuples which are in $[i_1]$ but not in $[i_2]$ projected onto the variables identified by the output specifier $[o]$.

(3) Atom Evaluation: This operator applies the (optional) method given by *meth* to each member of $[i_1] \times \ldots \times [i_n]$ creating the intermediate oid-stream $[i_1] \times \ldots \times [i_n] \times [res]$ where res is the result of the method application for each $i_1, \ldots, i_n$ combination. Next, the predicate *pred* is applied to the intermediate oid-stream and the result is projected onto those variables given in the output stream identifier $[o]$. More specifically:

- $[i_1], \ldots, [i_n]$ denote a set of oid-streams which represent the input to the object manager call. A variable name may appear in only one input stream.
- $[o]$ denotes the oid-stream which will be returned as output of the object manager call. A variable name may appear only once in the output stream. Variables referenced in the oid-stream $[o]$ are a subset of those in the input streams or the special identifier res.

- *meth* is an optional method application specifier of the form $<a, b, c, \cdots>.mname$, where $a, b, c, \cdots$ correspond either to variables in the input streams or are the textual representation of an atomic value. The special identifier *res* denotes the result of the method application and can be referenced in the output stream and predicate.
- *pred* is an optional predicate on objects in the input streams and/or result of the *meth* field. The full set of permissible predicates is given in Table 6. Variables in the predicate correspond either to variables in the input streams, the special identifier *res* or are the textual representation of an atomic value (denoted by *const* in the table).

Table 6: Predicates allowed in $\mathbf{OM}_{eval}$ calls.

$o_i == o_j$
$o_i \in o_j$
$o_i =_{\{\}} o_j$
$const = o$
$const \in o$
$o \in const$
$const =_{\{\}} o$

An $\mathbf{OM}_{eval}$ call must have either a method or a predicate specified, and can have both if required. If specified, the method is always applied before the predicate is evaluated. The special identifier *res* denotes the result of the method application and can be referenced in the output stream or predicate only if a method is specified.

The input streams may contain variables which are not referenced in the output stream, the method or the predicate. In this case the respective oids in the input streams are ignored. Variables referenced in the input streams and output stream but not in the method or predicate are carried through without modification. In this case, the unreferenced oid in each input tuple which satisfies the predicate after the optional method has been applied is copied unchanged to the corresponding output tuple. There is no relationship or restrictions on the ordering of variables in the input streams and output stream.

Example 7.1 Consider the atom evaluation operation

$$\mathbf{OM}_{eval}([a, b], [c], [res, c], <c, a>.m, b \in res)$$

The semantics of this operation are given by the following algorithm.

for (each tuple $t : <a, b, c> \in [a, b] \times [c]$) **begin** – iterate over cross product
  let *res* be the object returned by $<t.c, t.a>.m$[5] – method application
  if $(t.b \in res)$ **then** – set value inclusion
    add the tuple $<res, t.c>$ to the output stream
end $\diamond$

[5]We use the notation $t.c$ to denote component c of tuple t.

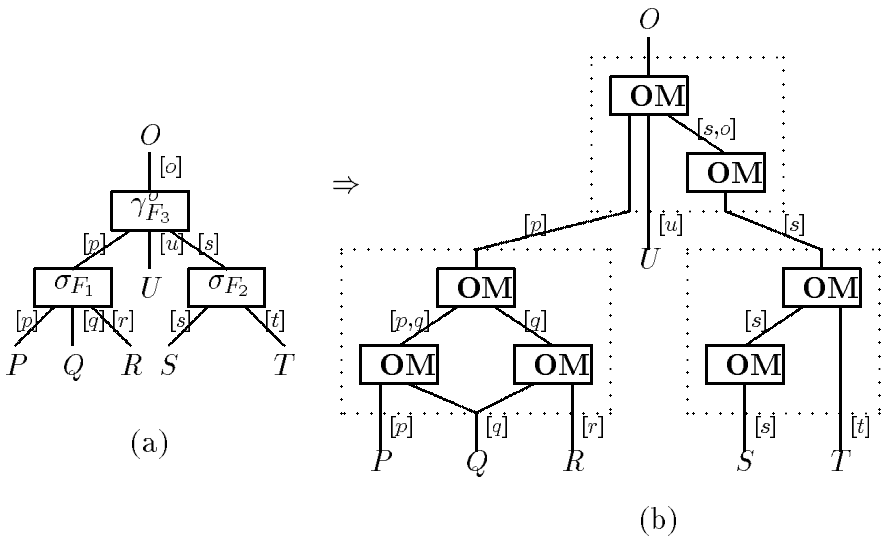

Figure 15: Mapping object algebra expression trees to object manager operation trees.

(4) Stream Reduction: This operator combines and reduces the number of input streams by performing an equijoin on those variables which are common to all input streams. This requires that all input streams have at least one variable name in common. The semantics of the operation is best described using an example.

Example 7.2 Consider the stream reduction

$$\mathbf{OM}_{\bowtie}([a,b,c],[b,d,c],[e,c,b],[a,b,e])$$

The variables common to all input streams are b and c. We can rewrite the operation as

$$\mathbf{OM}_{\bowtie}([a,b_1,c_1],[b_2,d,c_2],[e,c_3,b_3],[a,b,e])$$

in order to differentiate the different sources for variables b and c. The input streams are first combined by taking their cross product which results in the oid-stream $[a,b_1,c_1,b_2,d,c_2,e,c_3,b_3]$. The final result stream is of the form $[a,b,e]$ and contains only those tuples from the previous intermediate result where $(b_1 = b_2 = b_3) \wedge (c1 = c2 = c3)$. $\diamond$

7.2 Plan Generation

Execution plan generation can be thought of as creating a mapping from object algebra expression trees to trees of object manager operations. A query is initially represented as a tree of object algebra operators as shown in Figure 15(a). Edges in the figure have been annotated with oid-stream labels to indicate that a set of objects can be considered a stream of individual objects as well. For example, the set of objects denoted by P can be thought of as the stream of objects $[p]$ where $p \in P$. One unique feature of object algebra expression trees is that all edges represent streams of single objects, never streams of multiple objects. This is due to the closed nature of the algebra which insures that the output of any operation can be used as input to another.

The graph in Figure 15(b) represents an execution plan corresponding to the algebra tree on the left. An *execution plan graph* is a graph whose nodes are OM operators and whose edges are oid-streams. It is evaluated from the leaves to the root. The subtrees within dotted boxes are sequences

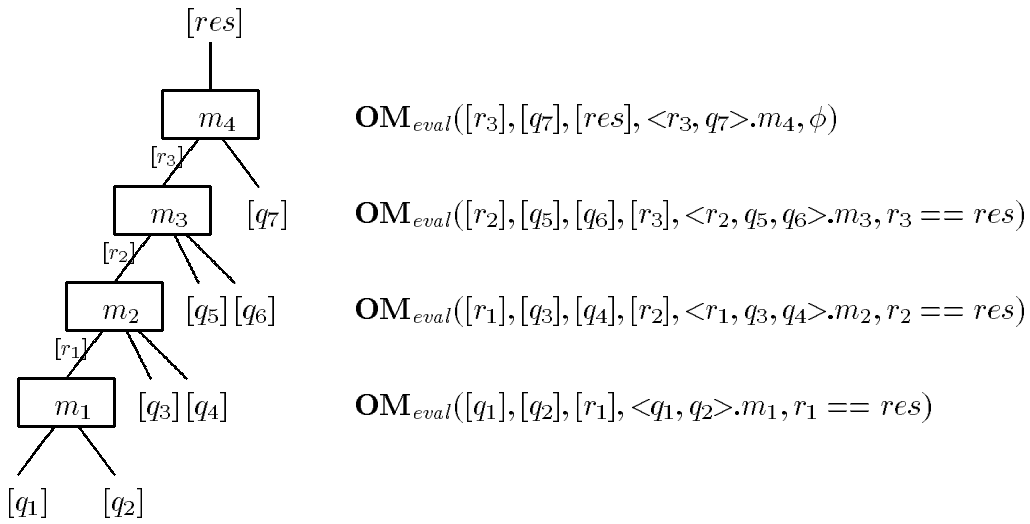

Figure 16: Execution plan generation for the object algebra map operator.

of object manager operations corresponding to individual algebra operators of the original query. Edges which do not cross subtree boundaries may represent streams of tuples of objects (e.g., $[p, q]$ and $[s, o]$). In addition, streams may be used as input to multiple object manager operations within a subtree, e.g., $[q]$.

The following sections shows how the mapping to object manager operators is performed for each of the object algebra operators ($\cup$, $-$, σ_F, $\mapsto_{mlist}$ and γ_F^t).

7.2.1 Union and Difference Operations

The union and difference operators map directly to their object manager counterparts. Inputs and output of these two algebra operations are always unary streams of objects even though $\mathbf{OM}_{\cup}$ and $\mathbf{OM}_{diff}$ accept streams of tuples of object identifiers.

7.2.2 Map Operation

Reviewing briefly, the map operator $Q_1 \mapsto_{m_1 \ldots m_n} \langle Q_2, \ldots, Q_k \rangle$ denotes the sequence of method applications $<q_1, \ldots, q_k>.m_1 \ldots m_n$ where $<q_1, \ldots, q_k>$ are drawn from $Q_1 \times \ldots \times Q_k$. Since the object manager interface can only apply one method per call, the method sequence must be decomposed into individual method applications. Determining which q_i are a parameter for a given m_j has been treated previously in [SÖ90d] and is not repeated here. Figure 16 depicts how the map operation $Q_1 \mapsto_{m_1.m_2.m_3.m_4} \langle Q_2, Q_3, Q_4, Q_5, Q_6, Q_7 \rangle$ is represented as a sequence of OM operations. The full algorithm to perform this transformation is given in [SÖ90a] and is omitted here due to space limitations.

7.2.3 Select and Generate

The select and generate operators introduce complexity into execution plan generation due to their use of predicates. At first it may appear that the two should be treated separately as the select operator returns a subset of an input set while the generate operator generates objects from those in the input sets. But from the perspective of low level execution plan creation, they are quite similar. Consider again the selection predicate of Equation 28. Even though the operation is a

selection, the predicate generates values for q. There is no inherent difference in complexity between predicates for selections and those for generate operations. The only real distinction between the two is that the target variable of a generate operation does not correspond to one of the input sets. The first requirement in creating select and generate execution plans is to rewrite the predicate such that each atom corresponds to just a single object manager call. Several substitutions are given in [SÖ90a] which insure that there is a one-to-one mapping between atoms in the predicate and object manager calls.

We outline a simple algorithm for mapping select and generate algebra operators to execution plan graphs. The algorithm takes three inputs: (1) a set of atoms corresponding to a simplified predicate, (2) a set of variable names identifying inputs to the object algebra operation, and (3) the name of the target variable. Output is an execution plan graph. The algorithm uses a hypergraph [Ber73] representation of the predicate. The hypergraph contains one node for each unique variable name referenced in the atoms of the predicate and is initialized with an edge for each atom of the predicate which covers all nodes corresponding to variables referenced in the atom. (Note that edges in a hypergraph define subsets of its nodes.) The nodes are marked as either red or green. A green node indicates that values for this variable exist, either because the variable ranges over one of the input sets or because an object manager call has generated values for it. A red marking indicates that values do not exist, i.e., the variable may not be used yet. The node markings are initialized to reflect the variables which represent inputs to the object algebra operation. The algorithm proceeds by successively placing into the execution plan graph $\mathbf{OM}_{eval}$ operations for atoms (hypergraph edges) until all atoms have been placed. An atom is eligible for placement in the execution plan graph if all the nodes in its corresponding edge are green, or only one node is red but it represents a variable whose values are generated by the atom. The complete algorithm is given in [SÖ90a].

Example 7.3 We apply the algorithm described above to produce an execution plan graph for the select operation whose predicate was given in (28). Figure 17 shows the initialized hypergraph with an edge for each atom in the predicate. Note that the node for q is red while all others are green indicating q does not range over an input set. Initially, both atoms $a2$ and $a3$ are eligible for placement because all but one node in their respective hypergraph edges are green and each atom generates values for the single red node. Atom $a1$ is ineligible at this point as it does not generate values for the red node. Let us assume atom $a3$ is chosen at random leading to placement of its corresponding object manager call (labeled as $a3$ in Figure 17) in the execution plan graph. After placing $a3$, q is colored green since values now exist for it and the edge for atom $a3$ is removed from the hypergraph. At this point, both remaining atoms are eligible for placement and we assume atom $a1$ is randomly chosen. The output oid-stream of the corresponding **OM** call is $[p, q]$ because (1) atom $a1$ overlaps with $a2$ on q, and (2) p is the target variable and needs to be retained for the final result. The algorithm terminates after placing the remaining atom, $a2$. $\diamond$.

The algorithm outlined in this section is quite limited in that it can only generate execution plans which are a linear sequences of $\mathbf{OM}_{eval}$ operations. Specifically:

- only one execution plan is generated,
- the ordering of multiple eligible OM operations is determined by random choice and does not allow a cost-based analysis of different orderings,
- object manager operations are never performed in parallel, and
- $\mathbf{OM}_{\bowtie}$ is not used to reduce intermediate oid-streams.

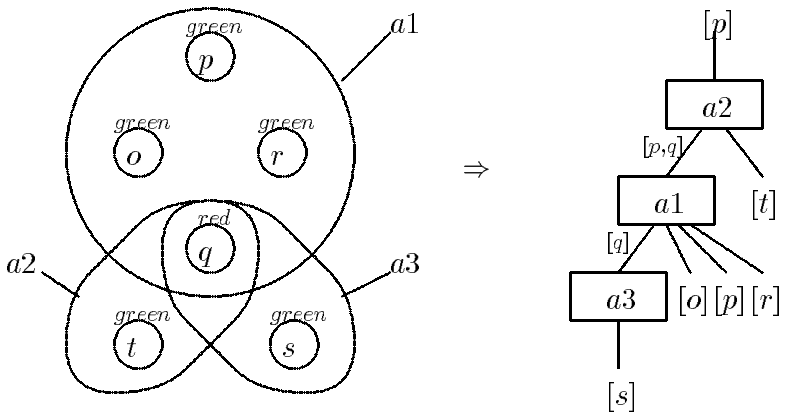

Figure 17: Hypergraph representation of a predicate and corresponding execution plan graph.

Ideally we would like to generate a family of execution plans from which a best plan can be chosen based on some cost criteria. An algorithm for accomplishing this is described in [SÖ91]. The algorithm uses *processing templates* which are extensions of *join templates* [RR82]. A processing template represents a family of logically equivalent execution plans. They are used as an intermediate formalism in mapping object algebra query trees to execution plan graphs.

8 Other Query Optimization Issues

There are a number of issues related to the approach that we have taken and related to the scope of our investigation that we would briefly like to touch upon. These issues involve the selection of the "optimum" execution plan and the optimization of the method executions.

8.1 Choosing the "Optimum" Plan

As mentioned above, the execution plan generation algorithm enumerates a processing template which identifies a family of logically equivalent query execution plans. Each connected subtree of edges in the processing template which includes all initial nodes and the final node is a valid plan. But which is the best plan?

Section 7.1.2 defined an object manager interface, but our research does not address its implementation. An implementation design would be highly dependent on the object representation, the technique used to bind method code to objects, and other system parameters. Thus, although we do not propose a specific cost function, we assume that the object manager is capable of using oid-stream statistics to derive a cost for calls to its interface.

Appropriate oid-stream statistics might be stream cardinality and information about the classes represented in the stream. For a given call, the object manager could derive a processing cost and statistics for the resulting output oid-stream. A processing template could then be annotated with cost information as follows.

Initially only leaf nodes (which are stream nodes) of the processing template would have stream statistics associated with them. If the leaf nodes correspond to the leaf nodes of the original object algebra query, then they represent the extent or deep extent of classes in the database and their statistics are readily available. Otherwise the leaf nodes represent the output of a previous subtree of object manager calls and the output oid-stream statistics of the appropriate subtree are attached.

Working from leaf to root in the processing template, the object manager cost function is used to assign a processing cost to each operator node as well as a set of stream statistics for the stream node the operator feeds into. All operator nodes and stream nodes in the processing template can be annotated with cost and statistical information in this fashion. The total cost of any specific execution plan within the processing template is the sum of the operator costs which are included in the execution plan's subgraph. If time information is included in the cost function, then when operator nodes execute in parallel, only the longest running operator should be included in the sum.

Note that cost information can not be used to prune the search space of the processing template generation algorithm. The search space of the algorithm is defined by the number of stream nodes present in the processing template at the start of each pass. This value can only be affected by the criteria used to define the "interesting permutations" which cause new operator and stream nodes to be created.

8.2 Optimization of method executions

Our research concentrates primarily on the optimization of query primitives. Ideally, query optimization should be possible for queries which utilize user defined methods. But this is highly dependent on the language used to define those methods. In the worst case, the only optimizations possible are those provided by the compiler of the method implementation language. Examples of such optimizations are inline subroutine expansion, removal of loop invariants, and efficient pipeline and register usage.

One approach assumes that behavioral abstraction is maintained at the logical level, while a structural object-oriented system exists at the lowest implementation level [GM88]. Objects and classes involved in a query are requested to *reveal* structural information by the query processor. Revealed expressions which still contain encapsulated behavior are recursively requested to reveal their equivalent (sequence of) structural expressions. When the revealing process bottoms out, the structural manipulation primitives are optimized by an extended relational query optimizer.

Another approach would be to use a purely functional language for user defined methods. Expressions in such languages can be recursively decomposed to sequences of primitive data manipulation operations. These decomposed sequences can then be optimized using the techniques described earlier.

Clearly, optimization of user defined methods is closely tied to the ability to reason about expressions in the method implementation language and is a significant area for future research.

8.3 Dynamic Schema Modification

One issue that typically complicates query processing is dynamic schema modification. The schema in an object-oriented database system varies according to the data model design decision discussed before. In our model which treats objects as instances of a single abstract data type whose extension is captured in one class, the schema consists of the class (type) lattice. However, different interpretations are possible as we discussed in earlier sections.

In our previous work and throughout this chapter, we assumed that the schema does not change during the execution of a query. In other words, once the query processor receives a query, the schema does not change until a response is retrieved. This may not be realistic, especially if the data model uniformly treats everything as an object. Thus, query processing has to co-exist with changes to the schema.

Dynamic schema modification has been studied within the context of object-oriented database systems. A comparative summary of this work is given in [NR89] which classsifies the types of shchema changes as follows:

- "changing class definitiions, i.e., instance variables or methods,
- modifying the class lattice by changing the relationships between classes, and
- adding or deleting classes in the lattice."

The interaction of these changes with query processing is a topic for future work.

8.4 Physical Optimization

This section is largely taken from Chapter 15 of [ÖV91]. Object storage and access is the responsibility of the object manager. In addition to providing a suitable interface for the generation of execution plans (as we discussed above), the object manager performs two other functions: physical clustering of objects, and localization of objects. In general, the object manager is also responsible for transaction management, but that's beyond the scope of our current discussion. Object clustering is the grouping of objects in the same memory extent, according to common properties, for example, the same value of an attribute or subobjects of the same object. By minimizing the number of memory extents to examine, fast access to clustered objects can be provided. Object localization gives the location of an object based on its identifier or content (e.g., an attribute value). It exploits object clustering information, possibly augmented with some form of indexing. The object manager, based on the clustering and localization of objects, has to provide efficient algorithms for implementing the interface operations.

As indicated before, relational databases can be used as object managers, but they are only efficient at managing simple objects. The problem is made significantly more difficult in object-oriented databases due to large atomic objects and complex objects. Large atomic objects are quite frequent in new database applications. For instance, a digitized image in an image database can require a few megabytes of storage. The object manager should be able to deliver only useful portions of a large atomic object to the application program or ADT operation that needs it. Complex objects may also be large because objects can be nested within each other using set and tuple constructors to an arbitrary degree. The typical example, from CAD applications, is a VLSI chip object that consists of several sections (e.g., 10), each consisting of many cells (e.g., 100), each containing more than 1000 transistors. Although the number of atomic objects of a VLSI chip (cells) is small (e.g., 100 bytes), the complex object may require several megabytes of storage. The object manager must be able to access an object and its subobjects rapidly if the entire complex object is needed. It must also provide efficient access to collections of subobjects without having to read the large complex object. The management of complex objects is also made difficult by object sharing, which permits each subobject to have more than one parent.

Storage techniques for relational databases may well be extended to support complex objects. The philosophy of this approach is to retain as much of the relational model and its underlying technology as possible. It applied initially to System R for CAD application support [LP83] and more recently to POSTGRES [SR86], an extension of INGRES. With the relational model, complex objects are decomposed into tuples (the subobjects). By treating tuple identifiers (TID) as attribute values, the object manager can maintain the links between the subobjects composing an object. An atomic object can be stored as a tuple ⟨TID, atomic value⟩. The nesting of a tuple t_1 within a tuple t_p is represented by storing the identifier of t_1 as an attribute of t_p. The nesting of a set of

tuples $\{t_1, t_2, \ldots, t_n\}$ within a tuple t_p can be represented by a binary relation containing the pairs $\langle$TID of t_p, TID of $t_i\rangle$, with $i = 1, \ldots, n$.

This storage approach brings out the benefits of the relational model. The access to subobjects stored in the same relation can be efficient if the clustering is appropriate. Furthermore, traditional indexing on attribute values is possible. Object query processing may be simplified with this approach. The conceptual query is first mapped into a relational query, which is expressed on the stored relations by replacing path expressions with the corresponding joins. The relational query can then be optimized using any relational query optimization technique. However, the disadvantage of this approach is that access to an entire complex object requires joins on tuple identifiers. Furthermore this approach is not sufficiently general since object identity is restricted to tuples and atomic objects. Therefore, sharing of set objects at the conceptual level is difficult to map at the physical level. Furthermore, whether or not all object-oriented query primitives may be mapped to relational ones efficiently is a question.

The alternative approach is to develop special complex object storage techniques. These must provide the capability of storing a complex object with its subobjects in the same memory extent. The early hierarchical and network database systems partially provided this capability. In CODASYL, restrictions are that a complex object must fit in a page and that records can only be shared using their physical identifiers (called database pointers). These techniques have recently been generalized to support nested relations and object-based models [KFC88]. Special attention has also been paid to the storage of atomic objects of arbitrary size. In EXODUS [CDRS86], an atomic object is a long byte sequence, which can be accessed in parts through a byte index. The storage of arbitrarily complex objects is more involved because of object sharing. The main difficulty of complex object storage with sharing is when the parent containing a shared object is deleted. In this case, the shared subobject must be relocated with another parent, which can be an expensive operation. A simpler solution would be to use the relational storage approach whenever objects are shared. Another difficulty with this approach is indexing in order to access entire objects or subobjects, since objects may be nested within other objects. A solution is to have path indexes [MS86] that associate attribute values with paths to the objects.

With complex object storage, complex objects may be mapped more directly at the physical level so that an object and its subobjects may be clustered in the same memory extent. In this case the conceptual query is mapped into a query expressed on the stored objects in the algebra for complex objects. The query processing algorithm could be similar to the exhaustive search approach by commuting all joins of stored objects, and for each one, selecting the best access method to the stored object. The only difference here is in the choice of the best access method in the complex object. Since the access to a complex object may involve path expressions and predicates on nested objects, the availability of path indexes is critical for efficiency.

9 Conclusions

In this chapter we discussed the issues that need to be considered in the development of query models and in the implementation of query processors in object-oriented database systems. The framework for the presentation is a query processing methodology that is depicted in Figure 1. One point that is evident from the foregoing discussion is the necessity for significantly more research in the development of query models and languages for object-oriented database systems. The ideas in this chapter are only preliminary points that arise from our work in this area. More work on query models and processing techniques is needed before a more definitive statement can be made.

Since we relied heavily on the methodology of Figure 1, it is important to comment on its

feasibility. The fundamental criticism has to be the linearity of processing. The methodology gives the impression that the steps can be followed one after the other to arrive at an execution plan which is "optimal." This certainly is not true. The transformation step will generate a number of different algebra trees and the plan generation step will produce more than one execution plan for each of these trees. It is important to note that a strategy has to be followed which cycles back and forth between the logical algebra optimization phase and the access plan generation phase. This would allow interleaving transformations which change the shape of the query with the introduction of access plan subtrees possibly resulting in more efficient plans.

One important area is to investigate how extensions to the data model can be integrated throughout the entire query processing methodology. For instance, our work has omitted recursive queries which are particularly important in knowledge base systems. Primitive operations to support recursion as well as others such as shallow and deep equality [KC86], additional predefined value types other than atomic, set or structural (e.g., tuple values [AGOP88]) or parametric types (e.g. *Set[t]* [SZ90]) would significantly enhance the usefulness of the model. Each addition to the basic data model must be propagated through the methodology of Figure 1. This means it must be incorporated into the calculus and algebra, type inference rules need to be developed, logical equivalences must be proven and the object manager interface must be extended. Performing this exercise for several extensions would provide insight into the tradeoffs between maintaining the proposed query processing methodology and completeness of the data model.

Improving the query languages is another important topic. The object calculus, while expressive, is not user friendly. Design of a user query language, perhaps an object SQL [Lyn88, Ont89], would enhance usability and uncover many programming language integration issues. The object algebra can be extended in two respects. The first is to provide support for object creating operations. This raises many philosophical as well as technical issues. For example, what is the class of an object created by such an operation and what methods are defined on it? Should such objects, and their new class, persist after execution of the query? The second extension to the object algebra involves support for universal quantification. This could be achieved by allowing quantification in predicates or by defining the algebra to operate on tuples of objects and providing a division operator similar to that of the relational algebra. Both approaches may affect the scope of transformations possible during logical optimization and the generation of access plans.

Designing an object manager implementation is another important area of research. Such a design must address many related issues such as object representation, physical partitioning of logical entities such as classes and their extents, object buffering, indexes, and how and when method code is bound to objects. The design also is affected by the underlying hardware architecture, e.g., uni-processor or multi-processor, and the available operating system services.

Acknowledgment

This chapter is partly based on a paper that the first author delivered as an invited talk [Özs91] and partly on the Ph.D. dissertation of the second author [Str91]. We would like to thank Vijay Kumar of the University of Missouri-Kansas City for the invitation to give the talk. We would also like to thank Jim Hoover, Li Yan Yuan and Warner Joerg of the University of Alberta and Sylvia Osborn of the University of Western Ontario for their careful reading of the dissertation and their many critical remarks. Lois Delcambre of the University of Southwestern Louisiana read an earlier version of this chapter and raised questions that led to significant improvements.

This research was supported in part by the National Sciences and Engineering Research Council (NSERC) of Canada under operating grant OGP-0951.

References

[AB84] S. Abiteboul and N. Bidoit. An algebra for non normalized relations. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 191–200, March 1984.

[ABD+89] M. Atkinson, F. Bancilhon, D.J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[AGOP88] A. Albano, F. Giannotti, R. Orsini, and D. Pedreschi. The type system of galileo. In M. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 101–119. Springer Verlag, 1988.

[AMY88] R. M. Akscyn, D. L. McCracken, and E. A. Yoder. KMS: a distributed hypermedia system for managing knowledge organizations. *Comm. of the ACM*, 31(7):820–835, July 1988.

[ASL89] A. Alashqur, S. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. 15th International Conference on Very Large Databases*, pages 433–442, 1989.

[Ast76] M. Astrahan. System r: A relational approach to data. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[AW86] R. Abarbanel and M. Williams. A relational representation for knowledge bases. In *Proc. 1st Int. Conf. on Expert Database Systems*, pages 191–206, 1986.

[Bee90] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, pages 353–382, 1990.

[Ber73] C. Berge. *Graphs and Hypergraphs*. North-Holland, 1973.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Eng.*, SE-13(1):65–76, January 1987.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *OOPSLA '86 Conference Proceedings*, pages 78–86, September 1986.

[BK86] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 53–59, 1986.

[BK90] F. Bancilhon and W. Kim. Object-oriented database systems: In transition. *Q. Bull. IEEE TC on Data Engineering*, 13(4):24–28, December 1990.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops, Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings*, pages 17–29, 1986.

[BKK88] J. Banerjee, W. Kim, and K. Kim. Queries in object-oriented databases. In *Proc. 4th Int. Conf. on Data Engineering*, pages 31–38, February 1988.

[Bro89] M.L. Brodie. Future intelligent information systems: AI and database technologies working together. In J. Mylopoulos and M.L. Brodie, editors, *Readings in Artificial Intelligence and Databases*, pages 623–640. Morgan Kaufmann, 1989.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer Verlag, 1984.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shetika. Object and file management in the EXODUS extensible database system. In *Proc. 12th International Conference on Very Large Databases*, pages 91–100, August 1986.

[CG88] B. Campbell and J. M. Goodman. HAM: A general purpose hypertext abstract machine. *Comm. of the ACM*, 31(7):856–861, July 1988.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (And never dared to ask). *IEEE Transactions on Knowledge and Data Eng.*, 1(1):146–166, March 1989.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 316–325, August 1984.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, June 1970.

[Con87] J. Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, September 1987.

[CW85] L. Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computer Surveys*, 17(4):471–522, December 1985.

[Dav90] K.C. Davis. *A Formal Foundation for Object-Oriented, Algebraic Query Processing.* PhD thesis, Center for Advanced Computer Studies, University of Southwestern Louisiana, 1990.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In *Proc. 2nd Int. Workshop on Database Programming Languages*, pages 80–102. Morgan Kaufmann, 1989.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database system. In *Proc. of the 2nd Int'l Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.

[DD91] K.C. Davis and L.M.L. Delcambre. A denotational approach to object-oriented query language definition. In *Proc. Int. Workshop on the Specifications of Database Systems.* Springer Verlag, July 1991.

[Dec86] D. Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *OOPSLA '86 Conference Proceedings*, pages 444–452, September 1986.

[DLA88] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds distributed operating system. In *Proc. 8th Int. Conf. on Distributed Computing Systems*, pages 2–17, June 1988.

[EE87] A. Ege and C. A. Ellis. Design and implementation of GORDION, an object base management system. In *Proc. 3th Int. Conf. on Data Engineering*, pages 226–234, May 1987.

[Fai88] J. Fairbairn. A new type-checker for a functional language. In M. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 69–87. Springer Verlag, 1988.

[Feu89] M. Feuche. Object oriented databases finding applications. *MIS Week*, April 10, 1989.

[Fis87] D. H. Fishman, et. al. Iris: An object-oriented database management systems. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Fre87] J. Freytag. A rule-based view of query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–180, 1987.

[Fro86] R. Frost. *Introduction to Knowledge Base Systems*. Macmillan, 1986.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 160–172, May 1987.

[GM88] G. Graefe and D. Maier. Query optimization in object-oriented database systems: The REVELATION project. Technical Report CS/E 88-025, Oregon Graduate Center, 1988.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The Language and its Implementation*. Addison Wesley, Reading, Mass., 1983.

[GV89] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases*. Addison Wesley, 1989.

[Hal88] F. G. Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Comm. of the ACM*, 31(7):836–852, July 1988.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 377–388, June 1989.

[HZ87] M. F. Hornick and S. B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987.

[HZ88] S. Heiler and S.B. Zdonik. FUGUE: A model for engineering information systems and other baroque applications. In *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, pages 195–210, 1988.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computer Surveys*, 16(2):112–152, June 1984.

[JS82] G. Jaeschke and H. Schek. Remarks on the algebra of non first normal form relations. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 124–137, 1982.

[Kae86] T. Kaehler. Virtual memory on a narrow machine for an object-oriented language. In *OOPSLA '86 Conference Proceedings*, pages 87–106, September 1986.

[KBC+88] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Integrating an object-oriented programming system with a database system. In *OOPSLA '88 Conference Proceedings*, pages 142–152, September 1988.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA '86 Conference Proceedings*, pages 406–416, September 1986.

[KFC88] S. Khoshafian, M.J. Franklin, and M.J. Carey. Storage management for persistent complex objects. Technical Report Technical Report ACA-ST-118-88, Microelectronics and Computer Corporation, 1988.

[Kim89] W. Kim. A model of queries for object-oriented databases. In *Proc. 15th International Conference on Very Large Databases*, pages 423–432, 1989.

[Kin89] R. King. My cat is object-oriented. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 23–30. Addison Wesley, 1989.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA '86 Conference Proceedings*, pages 214–223, 1986.

[LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Proc. IEEE Conf. Databases for Engineering Design Applications*, pages 115–121, May 1983.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 425–433, June 1988.

[Lyn88] P. Lyngbaek. Iris/OSQL 3.0 reference manual. Technical Report HPL-DTD-88-3, Hewlett-Packard Company, October 1988.

[Mai89] D. Maier. Why isn't there an object-oriented data model? Technical Report CS/E 89-002, Oregon Graduate Center, 1989.

[Man89a] F. Manola. An evaluation of object-oriented DBMS developments. Technical Report TM-0066-10-89-165, GTE Laboratories, October 1989.

[Man89b] F. Manola. Object-oriented knowledge bases, Parts I and II. *AI Expert*, pages 26–36 and 46–57, March and April 1989.

[MB90] F. Manola and A.P. Buchmann. A functional/relational object-oriented model for distributed object management – Preliminary description. Technical Report TM-0331-11-90-165, GTE Laboratories, December 1990.

[MD86] F. Manola and U. Dayal. PDM: An object-oriented data model. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 18–25, 1986.

[MG89] J. Marques and P. Guedes. Extending the operating system to support an object-oriented environment. In *OOPSLA '89 Conference Proceedings*, pages 113–122, October 1989.

[Min75] M.A. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Physchology of Computer Vision*. McGraw-Hill, New York, 1975.

[Moo86] D.A. Moon. Object-oriented programming with Flavors. In *OOPSLA '86 Conference Proceedings*, pages 1–8, 1986.

[Moo89] D.A. Moon. The COMMON LISP object-oriented programming language standard. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 49–78. Addison Wesley, 1989.

[MS86] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 171–182, September 1986.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *OOPSLA '86 Conference Proceedings*, pages 472–482, July 1986.

[MZO89] D. Maier, J. Zhu, and H. Ohkawa. Features of the TEDM object model. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 476–495, 1989.

[Nie89] O. Nierstrasz. A survey of object-oriented concepts. In W. Kim and F.H. Lochovshy, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3–21. Addison Wesley, 1989.

[NR89] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, pages 43–67, 1989.

[Ont89] Ontologic, Inc. *ONTOS SQL Command Reference*, November 1989.

[Osb88] S. L. Osborn. Identity, equality and query optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 346–351. Springer Verlag, 1988.

[ÖV91] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[OW89] G. Ozsoyoglu and H. Wang. A relational calculus with set operators, its safety, and equivalent graphical languages. *IEEE Transactions on Software Eng.*, SE-15(9):1038–1052, September 1989.

[Özs89] M.T. Özsu. From data management to knowledge management – Prospects for the next decade (invited paper). In *Proc. of CIPS National Congress*, pages 118–124, 1989.

[Özs91] M.T. Özsu. Query processing issues in object-oriented database systems – Preliminary ideas. In *Proc. 1991 Symp. on Applied Computing*, pages 312–324, 1991.

[PÖ91] R. Peters and M.T. Özsu. Formalization of a behavioral object data model. Technical Report in preparation, Department of Computing Science, University of Alberta, 1991.

[RR82] A. Rosenthal and D. Reiner. An architecture for query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 246–255, 1982.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, May 1979.

[SB85] M. Stefik and D. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, pages 40–62, 1985.

[SC88] B. R. Schatz and M. A. Caplinger. Searching in a hyperlibrary. In *Proc. 4th Int. Conf. on Data Engineering*, pages 188–197, February 1988.

[SH88] M. Stonebraker and M. Hearst. Future trends in expert data base systems. In *Proc. 2nd Int. Conf. on Expert Database Systems*, pages 3–20, 1988.

[SKY89] P.C. Sheu, R.L. Kashyap, and S. Yoo. Query optimization in object-oriented knowledge bases. *Data & Knowledge Engineering*, pages 285–302, 1988/89.

[SLU89] A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The treaty of Orlando. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 31–48. Addison Wesley, 1989.

[SÖ90a] D. Straube and M. T. Özsu. Access plan generation for an object algebra. Technical Report TR 90-20, Department of Computing Science, University of Alberta, June 1990.

[SÖ90b] D.D. Straube and M.T. Özsu. A model for OODB queries. In *Proc. Object-Oriented Database Task Group Workshop, NIST Technical Report NISTIR 4503*, pages 126–135, 1990.

[SÖ90c] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ90d] D.D. Straube and M.T. Özsu. Type consistency of queries in an object-oriented database system. In *Proc. ECOOP/OOPSLA '90 Conference*, pages 224–233, 1990.

[SÖ91] D.D. Straube and M.T. Özsu. Execution plan generation for an object-oriented data model. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, 1991.

[SR86] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 340–355, May 1986.

[SRL+90] M. Stonebraker, L.A. 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, September 1990.

[SS90] M. Scholl and H. Schek. A relational object model. In S. Abiteboul and P.C. Kanellakis, editors, *Proc. 3rd Int. Conf. on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 89–105. Springer Verlag, 1990.

[Ste87] L. A. Stein. Delegation is inheritance. In *OOPSLA '87 Conference Proceedings*, pages 138–146, 1987.

[Str91] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, University of Alberta, 1991.

[SZ90] G. Shaw and S. Zdonik. A query algebra for object-oriented databases. In *Proc. 6th Int. Conf. on Data Engineering*, pages 154–162, February 1990.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[US87] D. Ungar and R.B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–242, 1987.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 object manager: An overview. In *Proc. 15th International Conference on Very Large Databases*, pages 357–366, 1989.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation techniques of complex objects. In *Proc. 12th International Conference on Very Large Databases*, pages 101–110, August 1986.

[Wie84] G. Wiederhold. Knowledge and database management. *IEEE Software*, 1(1):63–73, 1984.

[Wie86] G. Wiederhold. Knowledge versus data. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 77–82. Springer Verlag, 1986.

[YO91] L. Yu and S.L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proc. 7th Int. Conf. on Data Engineering*, pages 670–677, 1991.

[Zdo86] S. Zdonik. Why properties are objects or some refinements of "is-a". In *ACM/IEEE Fall Joint Computer Conference*, pages 41–47, November 1986.

[Zdo88] S. B. Zdonik. Data abstraction and query optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 368–373. Springer Verlag, 1988.