Execution Plan Generation for an Object-Oriented Data Model*

Dave D. Straube† M. Tamer Özsu

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

Abstract

We address the generation of execution plans for object-oriented database queries. This is a challenging area of study because, unlike the relational algebra, a uniformly accepted set of object algebra operators has not been defined. Additionally, a standardized object manager interface analogous to storage manager interfaces of relational systems does not exist. We define the interface to an object manager whose operations are the executable elements of query execution plans. Parameters to the object manager interface are streams of tuples of object identifiers. The object manager can apply methods and simple predicates to the objects identified in a tuple. Two algorithms for generating execution plans for queries expressed in an object algebra are presented. The first algorithm runs quickly but may produce inefficient plans. The second algorithm enumerates all possible execution plans and presents them in an efficient, compact representation.

1 Introduction

There is significant interest in object-oriented database management systems (OODBMS) as an approach to handle the data management problems of complex application domains such as engineering databases, office information systems and knowledge bases. The specific features of an OODBMS are still topics of considerable debate [ABD+89, SRL+90] even though certain trends are emerging. However, what is not debated is that, in order to become a viable technology, object-oriented systems have to provide at least the data management functionality (e.g., declarative query formulation, optimization, transaction processing, etc.) that their relational counterparts provide. With this understanding, we have initiated a research project investigating query models and query processing issues in object-oriented database systems. This paper describes the results of one part of this investigation. Our companion papers [SÖ89, SÖ90b, SÖ90c] discuss other related issues.

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

†Present address: Banyan Systems, Inc., 115 Flanders Road, Westboro, MA 01581, USA; e-mail: daves@thing.banyan.com.

We have defined a query processing methodology for an OODBMS (Figure 1) similar to that for relational systems (see, for example [GV89, JK84]). Queries are expressed in a declarative language which requires no user knowledge of object implementations, execution paths or processing strategies. The query expression is first reduced to a normalized form and 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 classes in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested functions. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence preserving rewrite rules to the type consistent algebra expression. Lastly, an execution plan which specifies an ordering of primitive low-level operations while still respecting object encapsulation is generated from the optimized algebra expression.

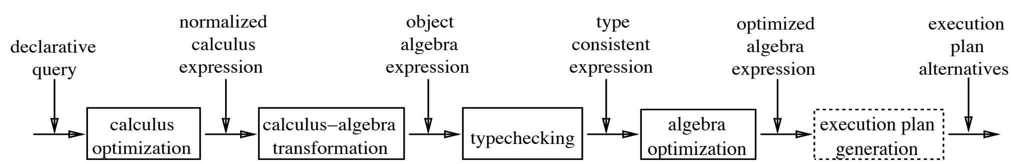


Figure 1: Query processing methodology

This paper addresses the last step in the query processing methodology shown in Figure 1, namely execution plan generation, which is the process of mapping high level representations of queries (i.e., object algebra expressions) to sequences of data manipulation operators of an object manager. Details of the data model, full definition of calculus and algebra, including translation algorithms, and some of the rewrite rules are covered in [SÖ90b]. The full suite of algebra rewrite rules is discussed in [SÖ89]. The type checking rules are given in [SÖ90c] and the typechecking algorithm is presented in [Str91].

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] and databases [CDRS86, CM84, EE87, HZ87, KBC+88, VBD89]. These object managers differ in terms of their support for data abstraction, concurrency, and object distribution. In addition, they are typically oriented towards "one-at-a-time" object execution which is an inefficient paradigm for query processing.

The fundamental contributions of this paper are the following:

- Definition of a new OM interface which maintains many features of previous object managers but operates on streams of objects. This definition would not have been necessary if there was a standard, widely-accepted OM interface. In the absence of such a standard, we define our own interface and use it in the generation of execution plans.
- Description of algorithms for generating execution plans whose processing steps are calls to the stream-oriented object manager interface.

Along these lines the paper is organized as follows. Section 2 reviews the object model and query language for which we generate query execution plans. Section 3 presents the object manager interface. Next, two algorithms for generating query execution plans are developed. The algorithm of Section 4 is simple but may not find best plans. Section 5 presents a more complex algorithm which finds all feasible plans. In Section 6 we discuss two issues related to query optimization in OODBMSs that we do not address specifically in this paper: the use of OM cost functions to select an "optimum" execution plan and the optimization of method executions. We conclude in Section 7 with some observations about our methodology and suggestions for future work.

2 Overview of the Data and Query Model

This section presents the fundamental features of the data model as well as the query model that we use to investigate query processing issues in object-oriented database systems. Due to space constraints, the description given here is brief and appeals to intuition. For a rigorous and formal definition of these concepts, the reader is referred to [SÖ90b] and [Str91]. As with the OM interface, the definition of the data model, which encompasses many of the features common to other object data models, is necessitated by the lack of a standard model specification.

2.1 Objects

Objects are viewed as instances of *abstract data types* (ADT) 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 primitive query operations. All other relations among objects are implemented by the ADT interfaces.

2.2 Classes and Methods

Our model interprets a *class* both as a definition of an ADT interface via *methods* and as a template 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 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].

2.3 Primitive Object Operations

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. Figure 2 illustrates the processing denoted by this operation when we assume that methods m_1 and m_m take three arguments each, and method m_2 takes 2 arguments. 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 $<r_{m-1}, o_{n-1}, o_n>$. The notation $<o_1 \ldots o_n>.mlist$ will be used when the list of method names is unimportant.

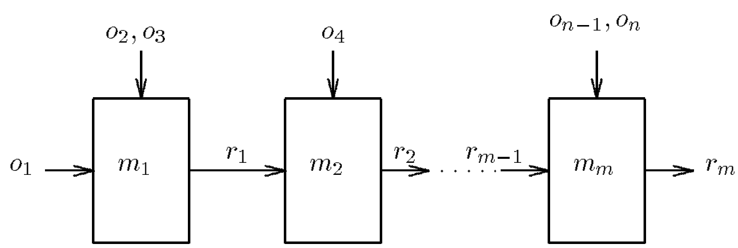


Figure 2: Composition of method applications.

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 (integer, 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.

We define four comparison operators which can be used in queries: $==$, $\in$, $=_{\{\}}$ and $=$ whose semantics are shown in Tables 1 and 2. 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.

2.4 Predicate Formation

Atoms are primitive operations of the data model which return a boolean result. Atoms reference lower case, single letter object variables which range over sets of objects when used in a query. The legal atoms are as follows:

Table 1: 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 2: 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

- $o_i \theta o_j$ 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 $=_{\{\}}$.

Predicates are formed by connecting atoms with $\wedge$, $\vee$ and $\neg$ as required.

Example 2.1 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 "59" the atomic value of the object denoted by p?

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"? ◇

2.5 Query Language – An Object Algebra

The object algebra contains both binary and n-ary operators. Let Θ be an operator in the algebra. We use the notation $P \; \Theta \; \langle Q_1 \ldots Q_k \rangle$ for algebra expressions where P and Q_i denote sets of objects. In the case of a binary operator we will use $P \; \Theta \; Q$ without loss of generality. The algebra defines five object preserving [SS90] operators: union, difference, select, generate and map. These are fundamental operators; others may be defined (e.g., intersection) for convenience in terms of these. Object preservation means that algebra operators return objects which exist in the database and do not create new objects. We have restricted our consideration to object-preserving algebras for two reasons. First, any OODBMS 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 utilising relationships defined by the type hierarchy or defined by ADT operations on objects is a fundamental requirement. Second, object-creating operations raise a number of issues which were not addressed in this research, such as the type of the created objects and the operations they support, the relationship between object creation and dynamic schema evolution, and so on.

We will use a sample database similar to that of [Kim89] depicted in Figure 3. Double lines represent subclass relationships and thin lines denote method signatures. For simplicity, only unary methods (e.g., *manufacturer: Vehicle* $\rightarrow$ *Company*) are used in the examples although real databases would take advantage of multiple argument methods (e.g., *employees*: Company* $\times$ *City* $\rightarrow$ *SetOfPerson*). Methods marked with an asterisk such as *employees** and *cars** return objects with set values.

The following are the precise definitions of the algebra operators that are supported in our model.

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 in 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 $\{p | P(p) \wedge Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge F(p, q_1, \ldots, q_k)\}$.

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.

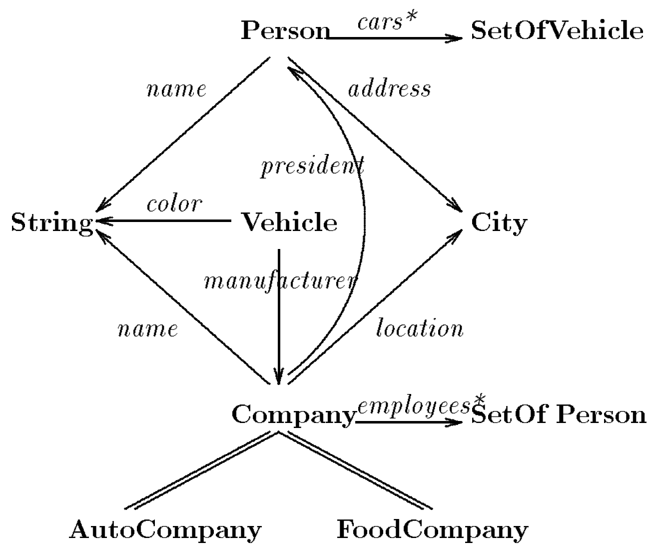


Figure 3: Sample database schema.

Example 2.2 The query "find all persons who live in a city which has an auto company" is an example of an explicit join. The select expression for this query is $Person\ \sigma_F\ \langle AutoCompany \rangle$ where $F \equiv (<p>.address == <a>.location)$, a ranges over $AutoCompany$, and p ranges over $Person$. $\diamond$

The result of this expression is a set of $Person$ objects, rather than sets of $< Person, AutoCompany >$ objects. This is due to the object-preserving nature of the algebra which disallows creation of new objects. In this sense, select is more like the traditional semijoin 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, i.e., 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 $\{t | Q_1(q_1) \wedge \ldots \wedge Q_k(Q_k) \wedge F(t, q_1 \ldots q_k))\}$.

Generating atoms are unique in that they generate values for variables which do not range over an input set of the query (Table 3). They are called generating atoms because they generate objects for x from a constant value (entry 5), from the content of other objects (entries 2,4), or by applying methods to objects (entries 3,4). As an illustration, consider the query $Q\ \gamma^p_{(p \in <q,r>.mlist)}\ \langle R \rangle$. Variables q and r in the predicate range over the argument sets Q and R respectively and thus can be considered 'bound' in the query. However, variable p is not bound to any argument set and the atom $p \in <q, r>.mlist$ will evaluate to true only when p ranges over the objects in the set value of the objects obtained by the method applications. Under these conditions then, the atom generates values for p.

Table 3: Generating atoms for x.

1	$x == o$
2	$x \in o$
3	$x == <o_1, \ldots, o_n>.mlist$
4	$x \in <o_1, \ldots, o_n>.mlist$
5	$x = a$

Example 2.3 An example is the query "return all cars driven by presidents of auto companies" where the *cars** method applied to a company president returns an object whose value is a set of car objects. The generate expression for this query is $AutoCompany\ \gamma_F^t\ \langle\ \rangle$ where $F \equiv (t \in <a>.president.cars)$. The query "find all cities auto company employees live in" combines unnesting of set values with method application. The algebra expression is $AutoCompany\ \gamma_F^t\ \langle\ \rangle$ where $F \equiv (x \in <a>.employees \wedge t\ == <x>.address)$. Note that predicate F contains generating atoms for two variables, x and t, although only objects for t are included in the result as specified by the γ_F^t notation. $\diamond$

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].

3 The Object Manager

As we indicated in Section 1, relational DBMSs benefit from the close correspondence between the relational algebra operations and low level access primitives of the physical system. Therefore, access plan generation in relational systems basically concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In OODBMSs the issue is more complicated due to the difference in the abstraction levels of behaviorally defined objects and their storage. Encapsulation of objects, which hides their implementation details, and the 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.

3.1 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. We, therefore, split what is usually called "access path selection" in relational systems into two steps: (1) *execution plan generation*, which is the mapping of object algebra expressions to object manager interface expressions; and (2) *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 database systems [ÖV91] which involves both global plan generation and local optimization. In this paper we are mainly concerned with the first step, briefly touching upon the second in Section 6.1.

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 $P\ \sigma_F\ \langle O, R, S, T\rangle$

$$F \;\equiv\; o == (<p,q,r>.m_1) \;\wedge\; (q \in t) \;\wedge\; (q ==<s>.m_2) \tag{1}$$

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

$$\{o | F(o,p,q,r,s,t) \text{ is true for } <o,p,r,s,t> \in\; O \times P \times R \times S \times T\;\} \tag{2}$$

	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 4: Dependencies between variables in a predicate.

Table 4 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 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.

3.2 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	$\mathbf{OM}_{eval}([i_1], \ldots, [i_n], [o], \mathit{meth}, \mathit{pred})$	– 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 $<a, b, \cdots>.mname$, where $a, b, \cdots$ 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 5. 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 5: 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 3.1 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 ${<}t.c, t.a{>}.m$[1] – method application
  if ($t.b \in res$) **then** – set value inclusion
    add the tuple ${<}res, t.c{>}$ to the output stream
end ◇

(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 3.2 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 $(b_1 = b_2 = b_3) \wedge (c1 = c2 = c3)$. ◇

4 Execution 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 4(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.

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

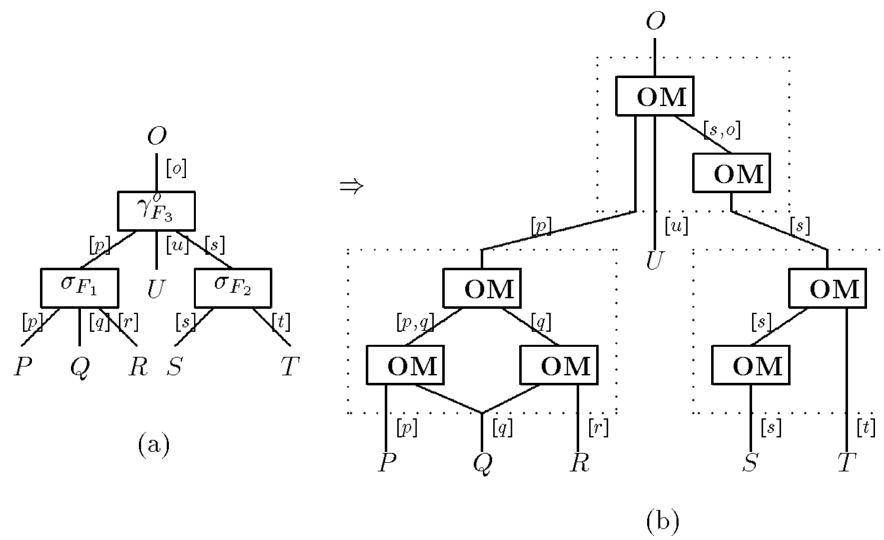


Figure 4: Mapping object algebra expression trees to object manager operation trees.

The graph in Figure 4(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 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., $[q]$.

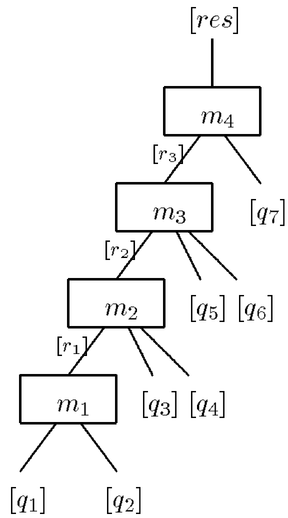
The following sections shows how the mapping to object manager operators is performed for each of the object algebra operators ($\cup$, $-$, σ_F, $\mapsto_{mlist}$ and γ^t_F).

4.1 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.

4.2 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 parameters for a given m_j is discussed in [SÖ90c] and is not repeated here. Figure 5 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.



$$\mathbf{OM}_{eval}([r_3], [q_7], [res], <r_3, q_7>.m_4, \phi)$$

$$\mathbf{OM}_{eval}([r_2], [q_5], [q_6], [r_3], <r_2, q_5, q_6>.m_3, r_3 == res)$$

$$\mathbf{OM}_{eval}([r_1], [q_3], [q_4], [r_2], <r_1, q_3, q_4>.m_2, r_2 == res)$$

$$\mathbf{OM}_{eval}([q_1], [q_2], [r_1], <q_1, q_2>.m_1, r_1 == res)$$

Figure 5: Execution plan generation for the object algebra map operator.

4.3 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 1. 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 4.1 We apply the algorithm described above to produce an execution plan graph for the select operation whose predicate was given in (1). Figure 6 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 6) 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, $a2$. $\diamond$

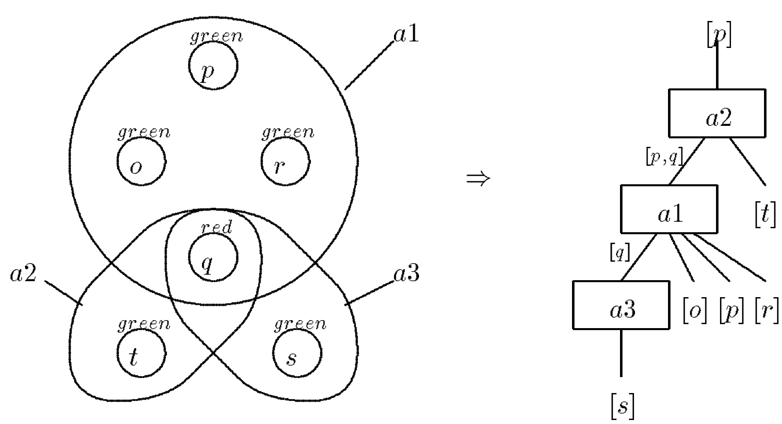


Figure 6: Hypergraph representation of a predicate and corresponding execution plan graph.

5 Select/Generate Execution Plans Revisited

The previous section introduced the notion of an execution plan as a tree of object manager operations. Queries expressed as trees of object algebra operators are converted to execution plans by mapping each operator in the algebra tree to a corresponding subgraph of object manager operations. Outlines of algorithms were given to perform this mapping for union and difference, map, and the select and generate operators.

This section examines the mapping process for select and generate operators in more detail. The algorithm presented in Section 4.3 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.

Ideally we would like to generate a family of execution plans from which a best plan can be chosen based on some cost criteria. To assist us in an exhaustive generation of execution plans, we extend the notion of a *join template* [RR82] to define a *processing template*. 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. A processing template for the predicate of Equation 1 is given in Figure 7.

A processing template consists of two types of nodes: *stream nodes* and *operator nodes*. Stream nodes (drawn as rectangles in Figure 7) represent intermediate results in a tree of object manager operations, i.e., execution plan graph. In other words, stream nodes reflect the variables present in an intermediate oid-stream and the atoms which were evaluated to produce them. Since there are conceivably many ways to produce equivalent oid-streams, each stream node in the processing template represents an *equivalence class* of oid-streams.

Each stream node has two fields. The top field denotes the object variables present in the oid-stream. The bottom field denotes which atoms have been evaluated in order to create the oid-stream, but does not indicate the order in which the atoms were evaluated. We will refer to these atoms as being *consumed* by the stream node.

Operator nodes (drawn as circles in Figure 7) denote the $\mathbf{OM}_{eval}$ or $\mathbf{OM}_{\bowtie}$ operations in a execution plan graph. An operator node is labeled with an atom number ($a1$, $a2$, etc.) if it corresponds to a $\mathbf{OM}_{eval}$ operation and with $\bowtie$ if it is a stream reduction operation.

Stream nodes with no consumed atoms, i.e., the leaf nodes, represent the original input streams of an object algebra select or generate operator. We define the *final node* as the stream node in the processing template whose variables field contains just the target variable of the object algebra operator and whose atoms consumed field contains all the atoms in the object algebra operator's simplified predicate. The final node is always node 0.

Edges represent the flow of tuples from one operator node to the next. Referring to Figure 7, nodes 1 through 5 represent the original input streams to the algebra operation of Equation 1 and node 0 represents the final result. Node 6 is the result of the object manager operation $\mathbf{OM}_{eval}([s], [q], {<}s{>}.m_2, q == res)$ and node 7 is the result of $\mathbf{OM}_{eval}([t], [q], \phi, q \in t)$. Each of these stream nodes represents an equivalence class of

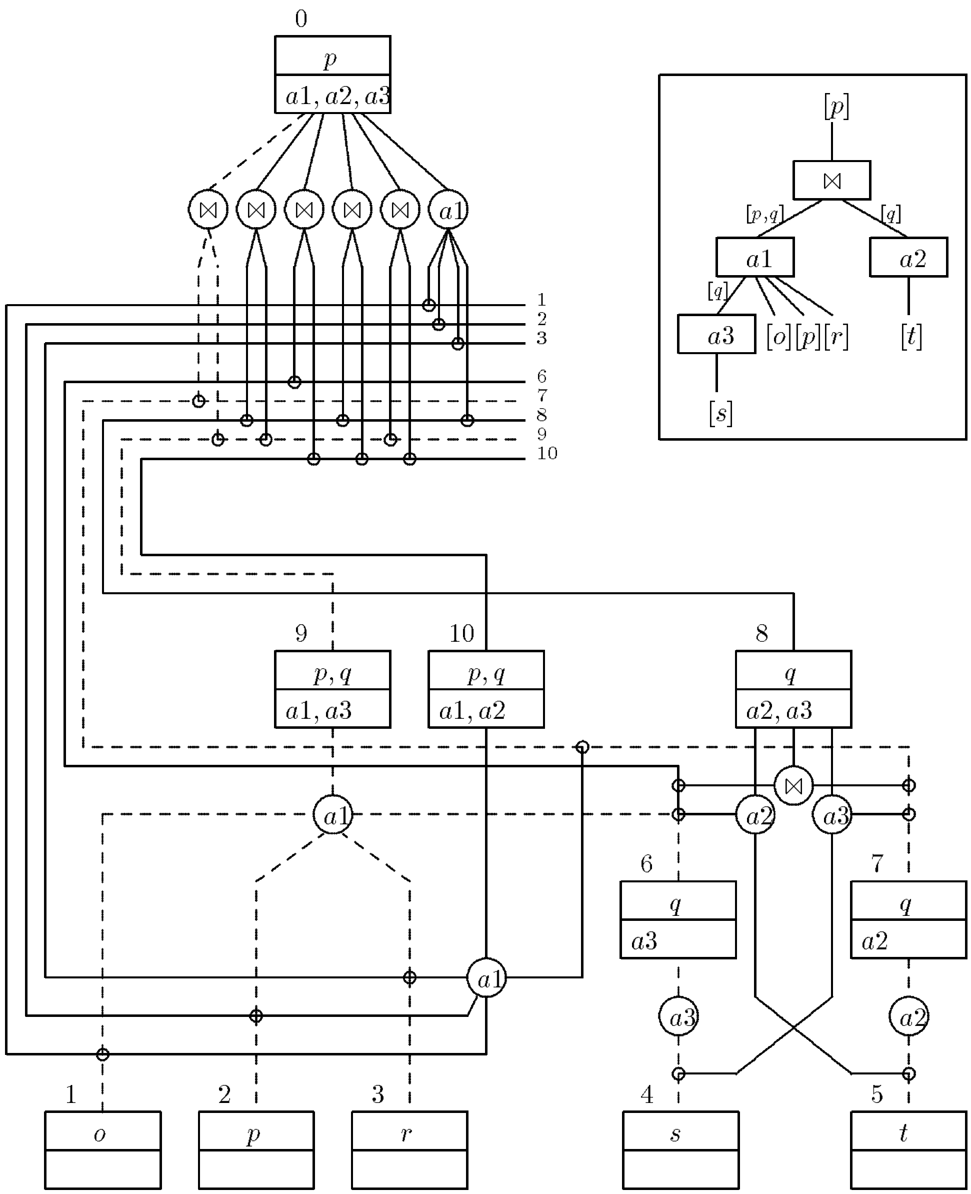


Figure 7: Processing template and one of its execution plans.

size one as they each only have one operator node feeding into them. Node 8 represents an equivalence class with three members. Using oid-streams subscripted with their processing template node numbers to indicate their source, the following OM calls all create the equivalent output denoted by stream node 8.

$$\begin{array}{ll}\mathbf{OM}_{eval}([q]_6, [t]_5, [q]_8, \phi, q \in t) & \text{(atom } a2\text{)} \\ \mathbf{OM}_{eval}([q]_7, [s]_4, [q]_8, {<}s{>}.m_2, q == res) & \text{(atom } a3\text{)} \\ \mathbf{OM}_{\bowtie}([q]_6, [q]_7, [q]_8) & \text{(reduction on } q\text{)}\end{array}$$

Each connected subtree of edges in the processing template which includes all initial nodes and the final node is a valid execution plan. As an example, the dashed edges in Figure 7 correspond to the execution plan shown in the top right of the diagram.

The full algorithm which, given a select or generate operation in the object algebra, returns a processing template which enumerates all possible execution plan graphs for that operation, is presented in [SÖ90a]. This paper outlines the algorithm using an extended example (Figure 7).

This example shows how a processing template is developed for the object algebra operation of Equation 1. An initial processing template is created by identifying the input streams of the algebra operation and placing nodes for each of them. In this example the input streams are $[o]$, $[p]$, $[r]$, $[s]$ and $[t]$ corresponding to nodes 1,2,3,4 and 5 respectively. The final node, node 0, is also placed in the processing template. Its variables field contains either the variable being restricted in the case of a select operation or the target variable in the case of a generate operation. The example operation is a selection on the input set P, thus p is placed in the final node. Similarly, all atoms of the reduced predicate (a1,a2,a3) are placed in the final node's consumed atoms field.

Once the initial processing template is created, the following steps are repeated until it is no longer possible to create any new stream nodes. Each iteration of the following steps is referred to as a *pass* through the algorithm.

Pass 1: Recall that processing template stream nodes represent oid-streams which can be combined to evaluate atoms or to remove duplicates. The first step of each pass then, is to enumerate all possible ways of combining stream nodes. We use the algorithm given in [OL88] for join enumeration but modify it slightly such that it does not produce combinations where a stream node is combined with itself (self-join). The final node is not included in the enumeration. Enumeration of the initial processing template results in the following permutations of stream nodes. Each permutation is shown as a set of stream node numbers and the sets are organized by size.

```
1: {1} {2} {3} {4} {5}
2: {1,2} {1,3} {2,3} {1,4} {2,4} {3,4} {1,5} {2,5} {3,5} {4,5}
3: {1,2,3} {1,2,4} {1,3,4} {2,3,4} {1,2,5} {1,3,5} {2,3,5} {1,4,5}
   {2,4,5} {3,4,5}
4: {1,2,3,4} {1,2,3,5} {1,2,4,5} {1,3,4,5} {2,3,4,5}
5: {1,2,3,4,5}
```

Similar to the filtering process described in [OL88], each permutation is tested to determine whether it is a *useful* combination of stream nodes. Each permutation of stream nodes defines mappings to sets of variables and sets of consumed atoms. For example the permutation {1,2,5} defines the mapping shown in Figure 8. We define two interesting types of mappings:

1. The variable sets are disjoint and together, all the variables exactly match those required by an atom which has not been consumed by any of the nodes in the permutation. In other words, an unused atom can be consumed using exactly those streams represented by the nodes in the permutation. The set of atoms which can be consumed by the combination of streams in the permutation (i.e., eligible for placement) is shown in Figure 8.

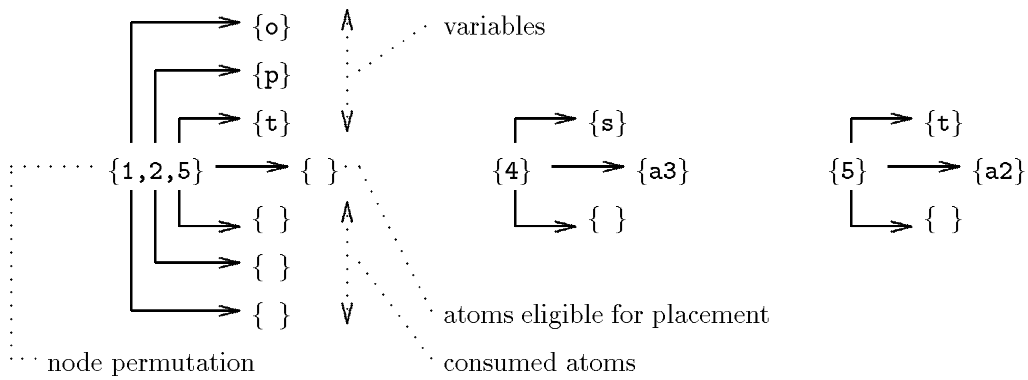


Figure 8: Mappings from stream nodes to variables, consumed atoms and eligible atoms.

   The first permutation in Figure 8, {1,2,5}, does not meet our criteria while the others do. Although the variables which {1,2,5} maps to are disjoint, they do not exactly match the variables required by an unconsumed atom.

   For each permutation with a non-empty set of eligible atoms, we consume each atom in the set by adding a stream node and operator node with appropriate connections to the processing template. In Pass 1, permutation {4} leads to the placement of stream node 6 and the operator node labeled a3 while permutation {5} leads to placement of stream node 7 and the operator node labeled a2. No other placements are possible. Since each stream node in the processing template represents an equivalence class of oid-streams, we do not always place a new stream node. If a stream node already exists with the appropriate set of variables and consumed atoms, only the operator node is added and the appropriate connections made.

2. One or more variables are replicated in each of the variable sets. Discussion of this case is deferred to Pass 2 since the condition does not occur during Pass 1 in this example.

At the end of the first pass the processing template consists of stream nodes 0–7 and the OM operations which connect them.

Pass 2: Pass 2 begins by again enumerating all possible combinations of stream nodes. However, since the contents of a stream node are not modified after it is initially added to the processing template (only new connections are made), we only need to enumerate all *new* permutations of stream nodes which were not considered in any of the previous passes.

As before, we build the mappings of stream node permutation to variables and consumed atoms and apply the filtering criteria. In this pass, both types of mappings which we consider interesting occur.

1. Mapping type 1 – the variable sets are disjoint and together, all variables exactly match those required by an atom which has not been consumed by any of the stream nodes in the permutation. These criteria are met by permutations {4,7}, {5,6}, {1,2,3,6} and {1,2,3,7}. Permutation {4,7}, which can be used to consume atom $a3$, would result in a stream node whose consumed atoms are $a2$ and $a3$ and whose variables field includes only q. Since this is identical to stream node 8, we just make the connections to node 8 rather than create a new stream node. This maintains the notion of a stream node representing an equivalence class of oid-streams. The same is true for permutation {5,6} which can be used to consume atom $a2$.

   Permutations {1,2,3,6} and {1,2,3,7} also meet our criteria and result in the creation of nodes 9 and 10 respectively.

2. Mapping type 2 – one or more variables are replicated in each of the variable sets. This condition means that several stream nodes exist with values for the same variable(s) and that an $\mathbf{OM}_{\bowtie}$ operation can be used to combine and reduce the oid-streams. Permutation {6,7} meets this criteria for variable q as shown in Figure 9.

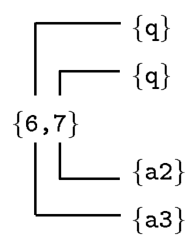


Figure 9: Node permutation with replicated variables.

   Each stream node in the permutation represents values for variable q generated by a different set of atoms. In other words, node 6 represents values for q generated by atom $a3$ while node 7 represents values for q generated by atom $a2$. The nodes are joined by an $\mathbf{OM}_{\bowtie}$ operation and all variables required by unconsumed atoms are carried through to the output oid-stream. In this case, the output oid-stream would contain only the variable q and would have consumed atoms $a2$ and $a3$. Since this is equivalent to node 8, we only add the $\mathbf{OM}_{\bowtie}$ operator node and make connections to node 8 rather than create an entirely new stream node.

Pass 3: Enumeration of all stream nodes results in the following interesting permutations: {1,2,3,8}, {7,9}, {8,9}, {6,10}, {8,10} and {9,10}. All of these permutations cause insertion of operator nodes only and do not cause any new stream nodes to be added to the processing template. The first permutation consumes an atom resulting in the placement of an $\mathbf{OM}_{eval}$ operation while all others result in $\mathbf{OM}_{\bowtie}$ operations. A further criteria is applied to the $\mathbf{OM}_{\bowtie}$ creating permutations which was not mentioned earlier.

Each of the stream nodes in the permutation must add to the consumed atoms field of the result. For example, permutation {`6,9`} is not acceptable as all of node 6's consumed atoms ({`a3`}), are already represented in those of node 9 ({`a1,a3`}).

The algorithm terminates after Pass 3 because no new stream nodes were created in this pass. In other words, enumerating all stream node combinations again will not result in any permutations which were not evaluated previously.

6 Discussion

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.

6.1 Choosing the "Optimum" Plan

Output of the enumeration algorithm described above is 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 3 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.

6.2 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.

7 Conclusion

This paper investigates the problem of generating execution plans for queries against object-oriented databases. Two primary topics are developed: (1) what interface should an object subsystem provide for efficient execution of queries, and (2) how to translate object algebra expressions into execution plans which consist of calls to the object subsystem interface.

Most of the object-oriented systems proposed to date provide the ability to apply methods and extract subcomponents for single objects only. However, processing queries efficiently requires that similar operations be applied to streams of objects. The object manager interface proposed in this paper does just this. Execution plans are represented as trees of object manager operations whose edges denote the flow of oid-streams.

The overall strategy for generating an execution plan consists of mapping each individual algebra operator in a tree of object algebra operators to a subtree of functionally equivalent object manager calls. Algorithms are outlined in this paper to perform this mapping for each of the five primary object algebra operators: union, difference, select, map and generate.

While the mapping process for the union, difference and map operators is primarily one-to-one, the mapping for the select and generate operations is one-to-many. A variant of the join template [RR82], called the *processing template*, is developed and used to represent the many subtrees of object manager calls which are logically equivalent to a single select or generate operation.

The execution plan generation method proposed in this paper introduces some interesting questions related to query optimization. Equivalence preserving transformation rules for logical optimization of object algebra expressions are derived in [SÖ90b]. The execution plan generation scheme proposed here assumes that these rules have been used to ameliorate the original object algebra query prior to plan generation. Plan generation then replaces each individual algebra operator with a "best" subtree of object manager calls. In other words, the overall shape of the query tree remains the same as that which was arrived at during logical optimization. One area of future research indicated by this methodology is the development of equivalence preserving rewrite rules for trees of object manager operations. Such rules would allow global optimization of the entire execution plan as opposed to merely picking "best" subtrees. Another interesting topic would be to develop an execution plan generation strategy which cycles back and forth between the logical algebra optimization phase and the execution plan generation phase. This would allow interleaving transformations which change the shape of the query with the introduction of execution plan subtrees possibly resulting in more efficient plans.

References

[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.

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

[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.

[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.

[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.

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

[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.

[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.

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

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

[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.

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

[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.

[OL88] K. Ono and G. Lohman. Extensible enumeration of feasible joins for relational query optimization. Technical Report RJ 6625 (63936), IBM Almaden Research Center, December 1988.

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

[SÖ89] D.D. Straube and M.T. Özsu. Query transformation rules for an object algebra. Technical Report TR 89-23, Department of Computing Science, University of Alberta, September 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. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ90c] 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.

[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.

[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.

[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.