Query Processing in the ObjectStore Database System

Jack Orenstein, Sam Haradhvala, Benson Margulies, Don Sakahara
Object Design, Inc.
{jack, sam, benson, don}@odi.com

Abstract

ObjectStore is an object-oriented database system supporting persistence orthogonal to type, transaction management, and associative queries. Collections are provided as objects. The data model is non-1NF, as objects may have embedded collections. Queries are integrated with the host language in the form of query operators whose operands are a collection and a predicate. The predicate may itself contain a (nested) query operating on an embedded collection. Indexes on paths may be added and removed dynamically. Collections, being treated as objects, may be referred to indirectly, e.g. through a by-reference argument. For this reason and others, multiple execution strategies are generated, and a final selection is made just prior to query execution. Nested queries can result in interleaved execution and strategy selection.

1. Introduction

ObjectStore is an object-oriented database system designed for use in non-traditional applications such as computer-aided design and manufacturing, computer-aided software-engineering, and geographic information systems. These will be referred to generically as CA*x* applications. CA*x* applications typically perform complex manipulations on large databases of objects with intricate structure. This structure is realized by inter-object references, (e.g., pointers from one object to another), involving hundreds of object types organized into type hierarchies. Objects are located by traversing these references and by associative queries. Objects may be updated, created, or deleted at any time.

CA*x* applications are written in general-purpose programming languages such as C and C++. Developers of these applications have traditionally been responsible for providing persistence for the values (or objects) manipulated. An object-oriented database system (ODB) is a good foundation for CA*x* applications because it provides persistence, transaction management, and other capabilities as described in [ATKI89].CA*x* applications interleave (what traditionally would be considered as) database operations and host-language operations at a fine level of granularity. For example, an application might traverse a sequence of nodes and arcs in a graph, and apply a function at each node. The traversal from one persistent object to another is a database operation, and the function to be applied is application-specific. If each database operation really resulted in communication between a client and a server, performance would be unacceptable. ObjectStore deals with this problem by moving database functionality into the client.

ODBs differ from relational (and extended relational) database systems in fundamental ways. The type system of an ODB is much closer to that of the host language, and the division of labor between client and server is completely different. For these reasons, the query language and query processor of an ODB are quite different from their relational counterparts. This paper describes the query language and query processor of ObjectStore.

Section 2 provides an overview of ObjectStore's architecture. Section 3 presents the programming interface to ObjectStore, focusing on collections and queries. In section 4, these features are compared with similar facilities in relational database systems. Query processing in ObjectStore is discussed in section 5. Section 6 discusses future plans for the ObjectStore query processor.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0403...$1.50

2. The architecture of ObjectStore

ObjectStore is based on a client/server architecture. The client requests pages from the server in response to page faults generated by the application. (By contrast, relational database systems send queries to the server.) A more complete discussion of ObjectStore's architecture is beyond the scope of this paper. See [LAMB91] for more information.

The ObjectStore server deals with pages only. Page contents are accessed by the client. In other words, only the client deals with objects. There is no reason in principle why the server could not understand objects. But based on our analysis of CA*x* applications requirements, we concluded that the performance requirements of these applications would best be met by doing data management tasks on the client, (as opposed to doing application-specific tasks on the server). Because collections are objects, they too are manipulated on the client, and this means that queries execute on the client. This does *not* mean that the entire contents of a collection or index is sent to the client in order to evaluate a query. As with other objects, only those pages containing referenced addresses are fetched from the server.

3. Collections, queries, and indexes

Queries can be expressed using an extended C++, supported by an extended C++ compiler, or through a function call belonging to a library interface. This section focuses primarily on interface. The library interface is discussed only briefly.

The discussion requires the introduction of some C++ terminology. Object classes are created using the *class* construct, (which is a generalization of the C `struct`). An object class consists of variable declarations, called *data members*, and function declarations, call *member functions*. Data members record the state of an object, and function members operate on that state. `C::d` refers to data member `d` of class `C`.

3.1. Collections

ObjectStore provides a library of collection classes which includes sets, bags, and lists. These classes support the usual operations, (set-theoretic operators, update, iteration, etc.). The design and implementation of this library is described in detail in [LAMB91]. For the purposes of this paper, only the set class is required. (All features to be described apply to all collection classes.) `os_Set<T*>` is a set of objects of type T. For example, a transient set of person objects, `people`, can be created as follows:

```
os_Set<person*> people;
```

In ObjectStore, persistence is orthogonal to type, so an os_Set<T*> can be either transient or persistent. To make people persistent in database db, the declaration would be:

```
persistent<db> os_Set<person*> people;
```

3.2. Schema

All examples in this paper will be based on the following schema, (expressed in C++):

```
class employee
{
public:
    string name;
    int age;
    employee* boss;
};

class task;
{
public:
    string description;
    int time;
    os_Set<employee*> team_members;
    employee* team_manager;
    os_Set<task*> successors;
    os_Set<task*> predecessors;
};

persistent<db> os_Set<task*> tasks;
persistent<db> os_Set<employee*>
employees;
```

This schema is used to model a set of related tasks. The tasks form a DAG through the `successors` and `predecessors` data members of `task`. `task::team_members` is the set of employees responsible for carrying out a task. The leader of the team is `task::team_manager`, (who belongs to `team_members`). Each task has a `description` and a `time`, (the time required to complete the task, measured in days). Each employee has a `name` and an `age`. `tasks` and `employees` are class extents.

3.3. Queries

An ObjectStore query is an expression specified by a query operator. A query expression evaluates to a collection, a single object, or a boolean. For example, the following query locates employees named Fred, and stores the result in a set:

```
os_Set<employee*>& freds =
    employees[: name == "Fred" :];
```

`[::]` is the query operator. It is preceded by a collection-valued expression, and brackets the selection expression. `[::]` returns a subset of the queried collection. In the example above, there is an implicit range variable of type `employee*`, matching the type of the set's members, so `name` refers to the name of one of the members of `employees`. The query returns a collection containing those member of `employees` satisfying the predicate.

Although the range variable is implicit, it can be referred to by the name `this`, (following a C++ convention for member functions). Thus the query above could be rewritten as `employees[: this->name == "Fred" :]`.

The selection predicate may be any expression of the host language, e.g.

```
employees[: age >= 30 && age <= 40 :]
```

locates people whose age is at least 30 and no more than 40. <=, >=, and && are C++ operators. The selection predicate may itself be another query. For example, the following query locates tasks with at least one team member younger than 20:

```
tasks[: team_members[: age < 20 :] :]
```

The nested query is existentially quantified, i.e. it returns true if there are any team members younger than 20.

In a nested query, each query has its own range variable named `this`. In some cases, it is necessary to refer to both inner and outer range variables. This is accomplished by introducing an "alias" for the outer range variable. For example, the following query locates those tasks that have a successor with the same manager:

```
task* t;
tasks[: t = this,    // establish alias
        successors[:
            team_manager ==
            t->team_manager :] :]
```

`t` is bound to the outer range variable. In the nested query, `team_manager` refers to the manager of the later task, and `t->team_manager` refers to the manager of the earlier. The inner predicate could also have been written as `this->team_manager == t->team_manager`. (The use of the comma in this query is standard C and C++, not part of the query language.)

ObjectStore queries are almost always over a single *top-level* collection, i.e. a collection not embedded in another object. In the examples above, `tasks` and `employees` are top-level collections. Due to the presence of embedded collections, which

can be viewed as non-1NF constructs, queries over multiple top-level collections are rare. However, they can be expressed. The following query finds tasks involving Fred:

```
tasks[:
    employees[: name == "Fred" &&
                task == task_id :] :]
```

This query, (which is based on a variation of the schema presented earlier), matches the `task` data member of an `employee` with the `task_id` data member of a `task`. Such a query is valid, but it reflects an approach to data modeling that is highly unusual among ObjectStore users, (and C and C++ programmers). Relationships are almost always denoted by direct reference rather than by matching values in unconnected objects. Furthermore, the presence of a `task_id` data member is very unlikely in an object-oriented schema, in which objects have inherent identity.

The ObjectStore query language is very similar to that of ORION [BANE88]. ORION has an explicit univeral quantifier, while ObjectStore does not, (negation can be used to accomplish the same thing). ORION defines an extent for each class and subclass, while ObjectStore does not — the user can declare any number of top-level collections. The essential feature of both languages is that of predicates over paths. Similar ideas appear in other object-oriented query languages [MAIE87, BLAK90], and even in a proposed SQL extension [COMM90] (using the dot notation of [ZANI83]).

3.4. Paths and Indexes

An index on a path can be created for a collection. All the paths for a given collection must originate in the same type of object, and the collection must store objects of this type. Example:

```
// Create some paths.
os_index_path time_path =
   pathof(task*, time);
os_index_path mgr_name_path =
   pathof(task*, term_manager->name);
os_index_path team_name_path =
   pathof(person*,
          team_members[]->name);

// Add indexes to a collection
tasks.add_index(time_path,
                os_collection::ordered);
tasks.add_index(mgr_name_path);
tasks.add_index(team_name_path);
```

`tasks` now has an ordered index on time, and unordered indexes on manager's names, and on team members' names, (`os_collection::unordered` could have been specified to achieve the same effect). Note that the last of these is a path through an embedded collection, i.e. the path is actually a set of paths.

Queries against any of these paths, or prefixes of these paths, will now be optimized. Examples:

```
// Use index on team_name_path.
tasks[:
   team_members[: name == "Fred" :] :]

// Use index on mgr_name_path
tasks[:
   team_manager &&
   team_manager->name == "Fred" :]

// Use prefix of index on
// mgr_name_path.
employee* fred = ...;
tasks[: team_manager == fred :]
```

The middle query above relies on "short-circuit" evaluation of predicates in C and C++. If `team_manager` is null, then the first clause is false, and there will be no attempt to compute `team_manager->name`.

There is no constraint on the type of the value at the end of the path. However, only the comparison operators are optimized. A user-supplied type may overload the comparison operators, (as long as the usual meanings are preserved), and the supplied definitions are used during index-based searches. This is the similar to what is proposed in [STON86].

3.5. Index maintenance

When there is a distinct sub-language for database access, as with SQL-based database systems, index maintenance is a simple problem. Database updates are a part of the sub-language, so it is known when an update requiring index maintenance has occurred. Because the update is performed on a named relation the database system knows which indexes require maintenance.

The situation is much more difficult for a system like ObjectStore, where there is no distinct sub-language. Any expression in the host language that updates a data member might require index maintenance. Suppose that `e` is an `employee*`, and consider this update:

```
e->age = e->age + 1;
```

All that is known here is that the age of some employee has been modified. It is not known how many collections contain `e`. Because collections are ordinary types, there may be collections containing `e` that are not class extents. Any or all of the collections that do contain `e` may have an index on age.

It is not feasible to check for indexes requiring maintenance every time a data member is updated. The performance consequences would be disastrous. For this reason, ObjectStore requires users to indicate those single-valued data members that could both serve as index keys and be updated following initialization. This is done by attaching the keyword `indexable` to the data member's declaration. Collections are inherently indexable. Here is an expanded version of the `employee` class:

```
class employee {
public:
   string name indexable;
   int age indexable;
   employee* boss indexable;
   const string social_security;
   float weight;
};
```

name, age, and boss are declared indexable, so updates to these data members are accompanied by index maintenance. social_security is declared to be constant following initialization, (this is standard C++), so index maintenance is not an issue. An index on social_security can be created, and it is known that updates due to changes in this data member will never occur. weight is neither const nor indexable. It can be changed at any time, and index maintenance is never required. Index paths terminating on this data member will not be permitted, and indexes on the data member cannot be created.

For multi-step paths, each single-valued, non-constant data member along the path must be declared indexable. For example, pathof(employee*, boss->name) would not be legal if boss or name was not declared indexable.

There are constraints on the use of indexable data members. These constraints guarantee that side-effects requiring index maintenance are detectable by the compiler. For example, the address of an indexable data member can only be taken if the user declares (using const) that updates will not be performed through the pointer.

3.6. The library interface

Queries can be expressed through the ObjectStore library interface. In its most general form, this is a three step process, 1) query creation, 2) binding of free variables, and 3) execution. The following query locates people whose ages are within a given range, [lo, hi]:

```
tasks[: time >= lo && time <= hi :]
```

lo and hi are free variables. The library interface version of the query can be created as follows:

```
os_coll_query& find_lo_hi =
os_coll_query::create(
      "task*",
      "time >= (int) lo && "
      "time <= (int) hi", db);
```

task* is specified as the type of the object in the collection being queried. The next argument is a string containing the query, with type specifications for the free variables. db is a variable naming a database in which schema information on the type task can be found.

The free variables are then bound:

```
os_bound_query find_5_10(
    find_lo_hi,
    (os_keyword_arg("lo", 5),
     os_keyword_arg("hi", 10)));
```

find_lo_hi is the query with free variables, and os_keyword_arg is a function that binds the free variables in the query to specific values.

The query can now be run against any collection containing person objects, e.g.

```
os_Set<task*>& needs_5_to_10_days =
                people.query(find_5_10);
```

4. Comparison with the relational model

Designing a conceptual schema for a relational database is very much like designing a set of types for a program written in an ordinary programming language. Differences in these activities are due to the different goals of relational databases and programming languages. The relational model emphasizes simplicity, conciseness, and theoretical soundness at the expense of generality. The set of primitive types and type constructors supported is limited, and third normal form schemas may not be intuitive, but within this domain, a very rich set of queries can be expressed concisely using the relational algebra, and freedom from certain "update anomalies" is guaranteed [DATE81].

General-purpose programming languages have very different goals. Generality cannot be sacrificed for theoretical principles. The data model, (i.e. the type system), must support semantically useful concepts. (For this reason, programming language type systems, especially those of object-oriented languages, have much in common with semantic and entity-relationship data models.) Conciseness is less important than efficiency, so high-level types such as relation or set are not provided as built-in types.

ObjectStore can be viewed as a persistent object-oriented language, extended with collection types[1] and associative queries. Because the queries are so closely integrated with the rest of the language, and because the underlying data model is semantic, not relational, ObjectStore queries are very different from relational queries, (this should be clear from section 3). Instead of relying on joins to reverse the effects of normalization, queries involve a small number of top-level collections, usually one, and the predicates involve paths originating in members of the collection.

This section compares the relational and ObjectStore data models, ending with a comparison of queries in the two models. Once the correspondences between relational and ObjectStore queries have been established, it will be easier to understand why query processing in ObjectStore differs from relational query processing.

4.1. Primitive types

The primitive types of the relational model include numbers and strings, and most vendors also support additional types such as date and time. The primitive types of ObjectStore are inherited from C++. These types include numeric types, a character type (which is actually numeric), and functions.

[1] via the ordinary mechanism for extensions — the object class.

4.2. Type constructors

The relational model has two declarative type constructors, tuple and relation. A relation is a set of tuples. Due to the requirement for first normal form, tuples can combine only primitive types, not tuples or relations. Types can also be generated dynamically, through projection and cartesian product.

The type constructors of ObjectStore, inherited from C++, include pointers, arrays, parameterized types, and classes. Multiple inheritance is supported, providing a concise way to define specialized types in terms of their differences from more general types. Type constructors can be combined in arbitrary ways. ObjectStore collections and relationships are defined using the class construct. There are currently no facilities for defining types dynamically.

4.3. Collections and type extents

A relation combines notions of type and extent. When a relation is defined, a tuple type, a relation type, and a set of tuples are created.

An ObjectStore type does not have any associated extent. Instead, there are collections, (defined using the class construct and parameterized types), and users may create any number of collections to store objects of a given type. Collections may be transient or persistent. The same object may simultaneously be present in any number of collections. If extents are desired, they can be specified concisely, although procedurally[2].

4.4. Normalization

The relational model requires first normal form relations. Good database design usually involves the use of higher normal forms. Normalization guarantees safety against "update anomalies", and ensures that information in the database can be retrieved with a single statement in a query language equal in power to the relational algebra. In spite of these advantages, properly normalized schemas are often non-intuitive, and as a result there are schema design tools that assist in the translation of entity-relationship (ER) diagrams into normalized schemas.

Even first normal form is alien to object-oriented databases, object-oriented programming, and to ordinary procedural programming. One of the goals of object-oriented programming is to facilitate the modeling of the real world, and normalization runs against this philosophy. Normalization rules seek to guarantee theoretical properties of the resulting schema, not facilitate modeling. Inter-object references, (i.e. pointers) and nested structures violate the requirements of first normal form, but are essential in programming. ObjectStore supports these constructs directly, simplifying the development of a working program from a conceptual schema.

4.5. Associative queries

Most relational queries are of the select/project/join variety. Joins re-establish connections between relations created during normalization, and projects discard attributes not required in the result. In the corresponding ObjectStore queries, nested queries and paths replace most join terms. For this reason, an ObjectStore query usually involves selection from one collection. The query returns a subset of the collection associated with the outermost query. This is analogous to a relational semi-join [ROTH80] in which the query returns all the attributes of one relation. However, this is not as restrictive as it first appears. Due to the lack of normalization, related objects can be located without further joins.

The following example demonstrates the relationship between relational joins and ObjectStore paths. Consider this ObjectStore query:

```
tasks[:
    team_manager &&
    team_manager->name == "Fred" :]
```

The relational schema corresponding to the ObjectStore schema of section 3.2 might be:

```
Employee(id, name, age)
Task(id, description, time,
     team_manager)
TaskMember(task, employee)
TaskSuccessor(task, successor)
TaskPredecessor(task, predecessor)
```

The SQL version of the query would is:

```
SELECT Task.id
FROM Task, Employee
WHERE Task.team_manager = Employee.id
AND Employee.name = "Fred"
```

The predicate `team_manager->name == "Fred"` in the ObjectStore query is expressed by the join and selection terms `Task.team_manager = Employee.id AND Employee.name = "Fred"` in the SQL query.

It could be argued that the target list should be `*` and not `Task.id`. The result of the ObjectStore query is an `os_Set<task*>`. From each `task` in the result, both `task` and `employee` information can be located without the need for another query. This is essentially what happens if the target list is `*`. Actually, the ObjectStore query also provides access to other objects reachable from `tasks` and `employees`. To do the same in SQL would require additional join terms.

A nested query involving an embedded collection translates to two join terms. For example, consider this ObjectStore query:

```
tasks[:
  successors[:
    team_manager->name == "Fred" :] :]
```

The SQL query corresponding to the above ObjectStore query is:

```
SELECT Earlier.id
FROM Task Earlier,
     TaskSuccessor,
     Task Later,
     Employee
WHERE Earlier.id = TaskSuccessor.task
AND TaskSuccessor.successor = Later.id
```

[2] A class extent can be maintained by the class's constructors and destructors.

```
AND Later.team_manager = Employee.id
AND Employee.name = "Fred"
```

The first two join terms are necessary to reverse the normalization that led to the creation of the `TaskSuccessor` relation. The third join term is required to connect `tasks` to their `team_managers`. The same information is recorded in the ObjectStore schema by a pointer.

5. Query processing and index maintenance

There is no concept of a class extent in ObjectStore. Instead, a program may create any number of collections, (including what amounts to a class extent, if desired), and bind them to variables, either directly, or indirectly using pointers. As a result, there is no direct association between queries and collections, and this affects the techniques that can be used for query optimization.

Consider the following piece of code:

```
foreach (task* t, tasks)
    f(t->team_members[: age < 20 :]);
```

(`foreach` is an ObjectStore construct for iterating over the members of a collection. It is similar to the foreach construct of Daplex [SHIP81].) For each task, `t`, the team members of `t` younger than 20 are located, and the set of these employees is passed to function `f`. A different collection is queried in each iteration of the `foreach` loop. The cardinalities of these collections may vary widely. Some of the collections may have an index on `age`, while others may not. It is therefore not possible, even in principle, to select one good strategy for this query before execution.

Consider a similar piece of code:

```
void g(os_Set<employee*>& s)
{
    f(s[: age < 20 :]);
}
...
foreach (task* t, tasks)
    g(t->team_members);
```

In this case, `t->team_members` is passed by reference to argument `s` of function `g`, and the query is associated with `s`. Here, it is not even known that an embedded collection is being queried. If any assumptions relevant for query optimization could be made about a collection due to the fact that it is embedded, they are of no use here.

Finally, consider this query:

```
tasks[: time > 100 :]
```

`tasks` is a specific collection in database `db`, (see section 3.2 for the declaration of `tasks`). However, even in this case, traditional optimization techniques are inapplicable since `db` could be bound to any database. Each database operated on by this application would have to have its own `tasks` variable.

These examples demonstrate that query optimization in ObjectStore cannot proceed as in a relational database system, because the same query can be applied to different collections. One approach used in some relational optimizers is to optimize the query the first time it is executed, and save the execution plan until the assumptions on which it is based, (e.g. availability of certain indexes), change. This doesn't work for ObjectStore since a query may be bound to different collections, possibly in different databases, even within a single run of an application. Clearly, some amount of optimization must be left until the moment that the collection to be queried is known. However, it is still possible to do a considerable amount of work when the query is translated.

The non-1NF data model of ODBs permits flexibility in indexing that is not possible in relational database systems. A relational schema would, due to normalization, require a relation corresponding to a nested collection in an ODB schema. For example, the `task::team_members` data member turns into the relation `TaskMember`. It is possible to index some `task::team_members` instances and not others, perhaps based on cardinality. But only a binary decision is required for the relation `TaskMember` — either there is an index or there isn't.

5.1. Overview of query processing

The major tasks involved in query processing are query analysis, code generation, strategy selection, and execution. There are actually two query processors, one in ObjectStore's extended C++ compiler, and another for use by queries submitted through the library interface. Both use the same runtime system for execution. The discussion focuses primarily on the compiler-based query processor. Section 5.5 discusses the library-based query processor.

A relational optimizer spends much of its time generating and evaluating strategies for computing joins. For each join, a variety of implementations can be considered, based on properties of the join attributes, (e.g., ordering, clustering, presence of indexes). Many of the joins in a query are used to connect one relation to another by joining a foreign key and a primary key. Navigating across n relations in this way requires $n-1$ joins, and the assortment of implementations for each join leads to a combinatorial explosion. Join optimization is less of a problem in object-oriented query languages for two reasons. First, joins between foreign keys and primary keys are replaced by paths, (essentially, pre-computed joins). Second, indexes may be created on whole paths (for some ODBs), and the optimization decision is to use the index or not. As a result, join optimization is a much simpler problem.

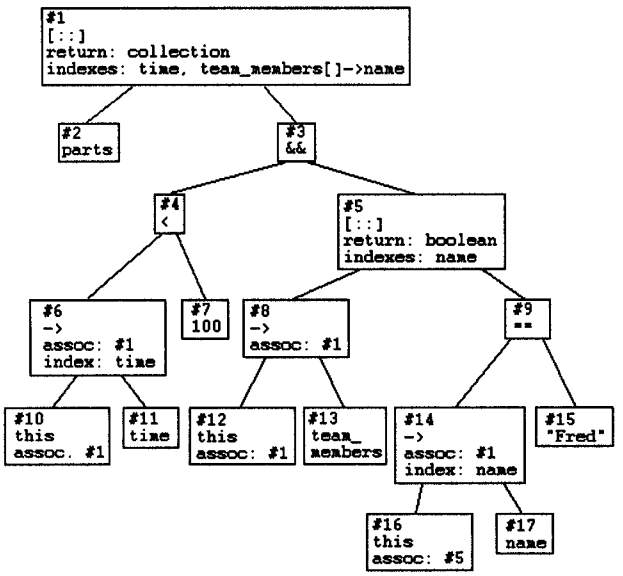
The discussion of query processing will trace this query through all stages of processing:

```
tasks[:
    time < 100 &&
    team_members[: name == "Fred" :] :]
```

5.2. Query analysis

The first step in query analysis is to create a parse tree representing the query. After parsing, information is propagated up and down the tree. There are seven node types in the parse tree, and the information propagated to the node depends on the

type, as described in the following subsections. The parse tree for the query is as follows:



5.2.1. Query nodes

The query operator is represented by the query node. The left subtree represents an expression returning a collection, and the right subtree represents an expression returning a boolean.

The query may return a subset of the collection being queried, one of the members (using a query operator not described in this paper), or a boolean. The return type is indicated by syntax and by the context of the query. The return type is useful information for optimization, as significant optimizations can be applied if the return value is a collection member or a boolean.

The set of index paths that could be used to evaluate any part of the query are propagated up from the right subtree. This information is used during code generation, to generate the strategy-selection code.

The parse tree above has two query nodes. #1 has two potentially useful indexes, on `time` and `team_members[]->name`, and #5 has one, on `name`.

5.2.2. Logical nodes

A logical node represents conjunction or disjunction, (see node #3). Information is propagated through but not captured in logical nodes.

5.2.3. Comparison nodes

Query optimization focuses on the usual comparison operators, =, <, etc. Each operand of a comparison is classified as a *path*, or as *garbage*. For example, in `employees[: name == "Fred" :]`, both operands of `==` are considered to be paths. The left operand is the path `this->name`. The value of this operand varies with the range variable of the query, so the query is said to be *associated* with the operand. The right operand is a literal with no associated query, but is also considered to be a *path*. In the parse tree above, nodes #4 and #9 are path/literal comparisons.

In a nested query, associated queries are used to determine which path is "more constant". Consider the following query, which finds tasks with the same manager as a successor task:

```
task* t;
tasks[:
    t = this,
    successors[: team_manager ==
                 t->team_manager :] :]
```

The left operand of `==` is associated with the inner query, and the right operand is associated with the outer query. The right operand is therefore constant for the inner query. So an index on `team_manager` for the embedded `successors` collections can be used to evaluate the inner query.

5.2.4. Variables

A variable is a free variable, a range variable, or a data member. A free variable is classified as *path* with no associated query, denoting a constant for purposes of optimization[3]. The associated query of a range variable is known from information propagated down the parse tree.

In the parse tree above, node #2 represents a free variable, nodes #10, #12, and #16 represent range variables, and nodes #11, #13, and #17 represent data members.

5.2.5. Side-effect free functions

A side-effect free function can be classified as *path* or *garbage*, based on its operands. Functions not known to be without side-effect shut off all query optimization. Currently, only one kind of function is understood as being side-effect free — a member function that does nothing other than return the value of a data member.

There are no such nodes in the parse tree above.

5.2.6. Path-forming operators

Consider the query `employees[: this->boss && this->boss->name == "Fred" :]`, (the range variable is made explicit). `->` is a path-forming operator. In the second clause of the query, it is used twice, to traverse from a collection member of type `employee*`, (represented by `this`), to the `boss` of the `employee`, to the `name` of the `boss`. If `boss` and `name` are both indexable, then an index on `boss->name` associated with the collection `employees` could exist, and be used to optimize the query. If either data member is not indexable then no such index could exist.

A node representing a path-forming operator propagates information up the parse tree. If the left subtree is labelled *path*

[3] The variable may change during the query, due to the invocation of a function with side-effects. However, the appearance of such a function in a query shuts off all optimization for that query.

and the right operand is an indexable data member[4], then the node is labelled *path*, and an index description is formed by appending the data member to an index description from the left operand. This index description is called a *potentially useful* index. In any other case, the node is labelled *garbage*. Assuming that `time` and `name` are indexable, nodes #6 and #14 are labelled *path*.

Nested query operators are also treated as path-forming. In node #5, the potentially useful index on `name` is propagated up to the nested query node. The path is extended to `team_members[]->name`, and is propagated up to the outermost query.

5.2.7. Other expressions

All other nodes are classified as *garbage*, and result in the suppression of index-based strategies.

5.3. Code generation

C++ is compiled into C, so the code generated to evaluate a query is a set of C functions. Logical and comparison nodes each give rise to a pair of functions, to deal with the presence or absence of indexes. These functions are called *compute* and *check*. The check function is used when no index is available. The collection being queried is scanned, and each member is passed to the check function where the predicate is evaluated. In terms of the parse tree, collection members are passed down the tree to logical and comparison nodes where the check functions are executed.

The compute function uses an index to evaluate a portion of a query. The index lookup is initiated at a comparison node by creating an object representing an index-based scan. Referring again to the parse tree, the scan object is passed upward. The node receiving the scan will "consume" it by using it in an iteration.

The following discussion will continue to refer to the parse tree, and scans or collection elements being passed up or down. It should be understood that when the generated code executes, the tree does not actually exist. Only variables and functions representing the nodes exist at runtime.

5.3.1. Query nodes

The code generated for a query node has two parts. First, a strategy is selected, and then execution of the selected strategy is initiated.

Strategy selection begins by noting the presence or absence of each index relevant to the query. (The collection being queried will be known when the strategy selection code is executed.) The potentially-useful indexes were propagated up the tree for this purpose. There may be any number of these indexes, yet a single decision is needed for the query node — whether to use an index-based strategy. The decision is made by propagating booleans representing the presence of each index up through the tree. For example, if either operand of a conjunction uses an index, then an index-based strategy can be used for the conjunction. Similar rules exist for other kinds of nodes.

If strategy selection indicates that an index-based strategy does not exist, then a scan is set up over the collection being queried, and each member is passed to the check function of the right child. If an index can be used, then control is passed to the compute function of the right child. Control keeps passing downward through compute functions until a compute function corresponding to a comparison node is reached. At this point, an index-based scan object is created and passed back up through the compute functions until it is consumed, (at which point another scan may be created).

5.3.2. Logical nodes

The compute function for a conjunction either uses two indexes, and intersects the results, or it uses an index for one operand and filters the result based on the other operand. In either case, the result is captured in a scan object which is returned to the caller. The compute function for a disjunction is used only when each operand has an index. The two results are combined into a new scan.

For both conjunctions and disjunctions, the check function simply invokes the check function for the two operands.

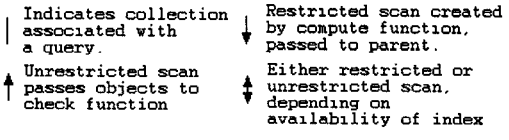
5.3.3. Comparison nodes

The compute function sets up and returns an index-based scan. The check function simply tests the object passed in.

5.4. Query execution

Query evaluation begins by checking to see what indexes are present. Execution for the entire query (except, possibly, nested queries) is planned by noting which clauses can be assisted by the use of an index. Currently, an index is used if it is present. This is not always optimal, e.g., if the collection's elements are clustered, or if the cardinality of the collection is sufficiently low. We do not currently use cardinality and clustering information to control the decision to use an index, although there is nothing to preclude us from doing so.

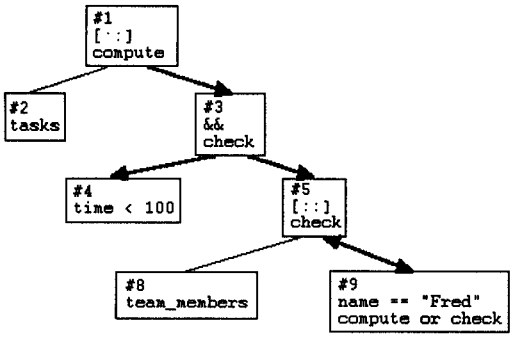
The possible execution strategies can be depicted graphically, in terms of parse trees, using the following legend:



In the diagrams that follow, the node identifiers refer to the parse tree of section 5.2.

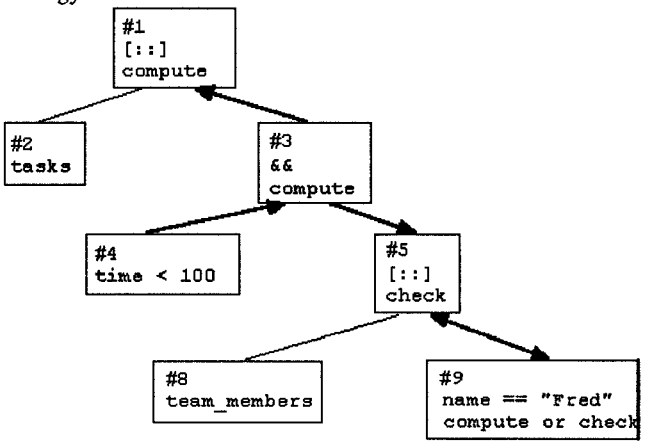
The simplest strategy does not use any indexes.

[4] To be precise, any data member that can support an index is acceptable. Specifically, the data member must be declare `const`, `indexable`, or it must be a collection.



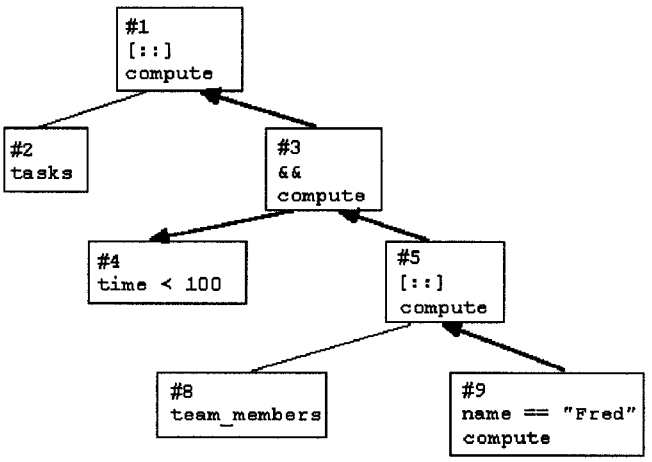
There are no indexes available for the outer query, so `tasks` is scanned in its entirety, and each task is passed to the check function of node #3. This function checks the restriction on `time`, and if it returns true, runs the nested query on that `task`'s `team_members`, using the check function. The double headed arrow below node #5 indicates that each set of `team_members` goes through strategy selection. If there is an index on `name`, it will be used to determine whether there is a `team_member` named Fred, and the compute function of node #9 will be used. If there is no index, an unrestricted scan is set up at node #5, and each name is examined by the check function of node #9. Because the query is existential, the scan can be ended as soon as one qualifying team member is found.

If there is an index on `time` (only) then the following strategy is used.



The compute function of node #3 will be used. This function sets up a restricted scan using the compute function for node #4 and passes qualifying `tasks` to the check function of node #5, which is processed as above.

If there is an index on `team_members[]->name` but not on `time`, the compute function of node #9 sets up a restricted scan on team member names. Unlike the previous strategies, embedded `team_members` sets are not handled individually, as the index is not on team member names, but is associated with `tasks` (not with a single task's team members). The result of this index lookup is a restricted scan that will return members of `tasks` having a team member named Fred. The scan is created at node #9 and passed up through node #5 to node #3. At node #3, an iteration over the scan is set up, and the returned parts are passed to the check function of node #4.



Finally, the last strategy is to use indexes on both time and team member names. The two restricted scans are carried out by invoking the compute functions for both sides of the conjunction, and the parts common to both are returned as the result of the query.

5.5. Query processing in the library interface

The basic tasks involved in query processing are the same when a library interface is used, however, the tasks are performed at different times. Query analysis is now performed at execution time when the query is defined via the functional interface. It is the user's responsibility to make sure that query analysis occurs at an appropriate time, (e.g., not inside time-critical loops). Strategy selection takes place immediately prior to execution of the query, when the collection being queried has been identified.

5.5.1. Query analysis

The analysis proceeds by first parsing the string representing the query into a tree representation. The nodes constituting this tree are conceptually similar to the nodes described in section 5.2, although their actual representation is different.

First, the tree is examined to determine all sub-expressions yielding values that are invariant with respect to enclosing query expressions. For example, in the query

```
task* t;
tasks[:
   t = this,
   successors[:
      team_manager == t->team_manager
      && age < 10 :] :]
```

the sub-expression `t->team_manager` is invariant with respect to the inner query, while `10` is invariant with respect to the entire query.

Next, sub-expressions that could form index paths are noted in the tree whenever they are used as an operand of a comparison operator whose second operand is an invariant sub-expression. Each such sub-expression, if it is evaluated unconditionally, is a candidate for a code-hoisting tree transformation. A code-hoisting transformation takes a sub-tree of the general form

```
<collection>[: ...
   && (<index path> == <invariant>)
   && ... :]
```

and transforms it into:

```
<collection>
   [: <index path> == <invariant> :]
   [: ... && true && ... :]
```

where <collection> is the original collection being queried, and <index path> and <invariant> are the index path and invariant sub-trees detected in the preceding steps of the analysis. The transformation yields a sequence of two queries, with the result of the first query feeding into the second. Indexes will be used to evaluate the first query, and the results will then be filtered by the second query which does not use any indexes.

5.5.2. Code generation

The annotated parse tree is transformed into one or more trees each reflecting an evaluation strategy for the query, given the availability of a specific set of indexes on the collections being queried. The transformed trees are generated by sequentially activating every combination of possible index paths in the parse tree. These transformed trees are the final result of query analysis, and are associated with the analyzed query for retrieval during query execution.

6. Future work

The work described here can be continued in a number of ways. Adding optimizations is a never-ending task. This includes handling different kinds of queries, but also expanding the scope of the query optimizer. For example, an intersection of two queries over the same collection can be optimized by using one query with an additional conjunction instead. The intersection is currently outside the scope of the query optimizer, so the optimization is not yet possible. Another possibility is to make the optimizer extensible, so that the set of optimizable predicates can be extended. A number of ObjectStore users have spatial applications, and they have asked if the optimizer could optimize overlap queries. Hard-wiring an overlap predicate into the optimizer and making use of a spatial index would be easy, but a more general mechanism is preferable.

An area of great importance to current and potential users is co-existence with relational databases. These users appreciate the advantages of having a single type system for transient and persistent data, and having associative queries integrated with the host language. But they still need to access data in "legacy systems" such as relational databases. We have therefore been investigating ways in which the ObjectStore and SQL schemas and query languages can be interrelated. (Some of these relationships were described in section 4.) This work will permit a variety of combinations: migration in either direction, either atomically or incrementally, ObjectStore queries to SQL databases, and SQL queries to an ObjectStore database as long as the ObjectStore objects can be described relationally.

References

ATKI89 Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. The object-oriented database manifesto. *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, (Kyoto Japan, December 1989).

BANE88 Banerjee, J., Kim, W., and Kim, K.-C. Queries in object-oriented databases. *Proceedings Fourth International Conference on Data Engineering*, (Los Angeles, 1988).

BLAK90 Blakely, J., Thompson, C., and Alashqur, A. Strawman reference model for object query languages. *Preliminary Proceedings of the First OODB Standardization Workshop*, Atlantic City, New Jersey (May 1990).

COMM90 Committee for Advanced DBMS Function. Third generation database system manifesto. *SIGMOD Record 19*, 3 (September 1990).

DATE81 Date, C. *An Introduction to Database Systems, 3rd edition.* Addison-Wesley, Reading, MA (1981).

LAMB91 Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Comm. ACM 34*, 10 (October 1991).

MAIE87* Maier, D., and Stein, J. Development and implementation of an object-oriented DBMS. *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds., MIT Press 1987.

ROTH80 Rothnie, J. *et al.* Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems 5*, 1 (March 1980).

SHIP81* Shipman, D. The functional data model and data language DAPLEX. *ACM Transactions on Database Systems 6*, 1 (March 1981).

STON86 Stonebraker, M. Inclusion of new types in relational data base systems. *Proc. 2nd International Conference on Data Engineering*, Los Angeles, CA (February 1986).

ZANI83* Zaniolo, C. The database language GEM, *Proc. ACM-SIGMOD Conference on Management of Data*, San Jose, CA (May 1983).

* Also in *Readings in Object-Oriented Database Systems*, S. B. Zdonik and D. Maier, Eds., Morgan Kaufmann, 1990.