Query Service Specification 11

11.1 Service Description

11.1.1 Overview

The Query Service provides query operations on collections of objects. The queries are predicate-based and may return collections of objects. They can be specified using object derivatives of SQL and/or other styles of object query languages, including direct manipulation query languages.

The term “query” has read-only connotations, but we use it to denote general manipulation operations including selection, insertion, updating and deletion on collections of objects. Throughout this chapter, the term “object” is used in the general sense to include data.

The Query Service can be used to return collections of objects that may be:

- Selected from source collections based on whether their member objects satisfy a given predicate.
- Produced by query evaluators based on the evaluation of a given predicate. These query evaluators may manage implicit collections of objects.

The source and result collections may be typed. The source collection may be specified by the client or may be the result of previous queries.

11.1.2 Design Principles

The Query Service exists to allow arbitrary users and objects to invoke queries on arbitrary collections of other objects. Such queries are declarative statements with predicates, including the ability to specify values of attributes; to invoke arbitrary operations; and to invoke arbitrary services within the OMG environment, such as the Life Cycle, Persistent Object, and Relationship Services.

To support the OMG architecture, the Query Service must allow querying against any objects, with arbitrary attributes and operations.

To be useful in practical situations, the Query Service must allow use of performance enhancing mechanisms, such as indexing.

To be useful in environments with database systems—object-oriented, relational, and other—and with other systems that store and access large collections of objects, the Query Service must map well to these native systems' internal mechanisms for specifying collections and using indexing. The Query Service must also allow the native systems to contribute to specifying collections and indexing.

To maximize usefulness to the community at large, the Query Service is based on existing standards for query and extended when necessary to accommodate other design principles.

The Query Service also supports flexibility in implementation and extensions.

11.1.3 Architecture

The Query Service design provides an architecture for a nested and federated service that can coordinate multiple nested query evaluators, much as the Transaction Service provides an architecture for a nested and federated service that can coordinate multiple nested resources managers.

Query Evaluators: Nesting and Federation

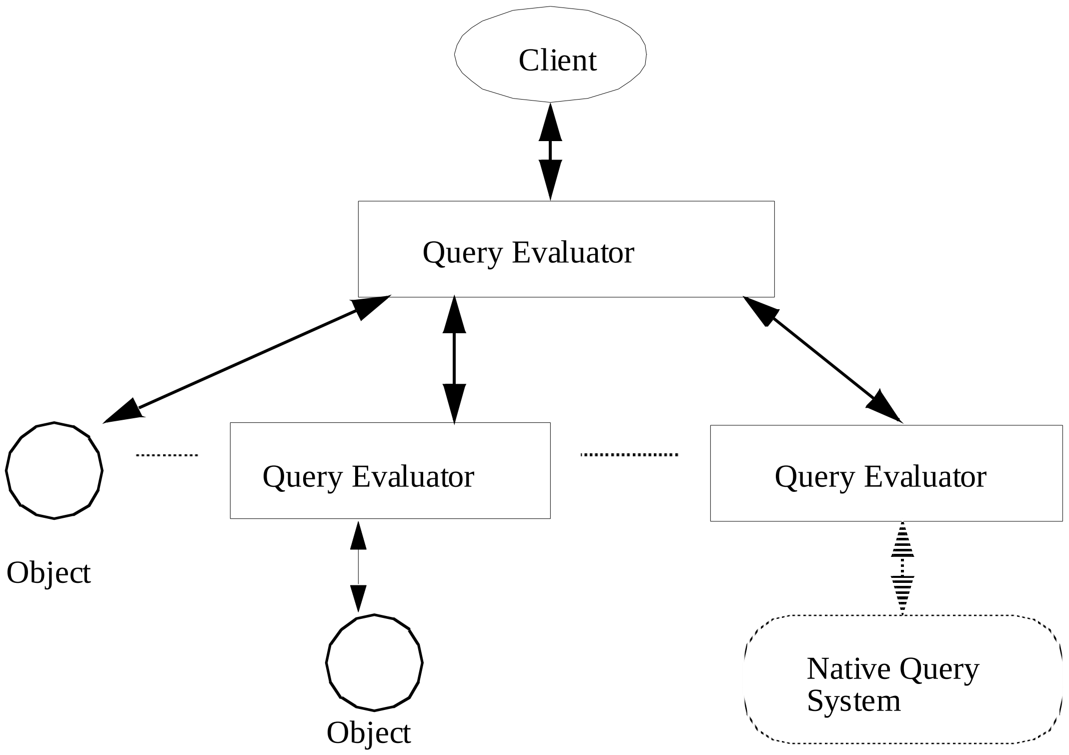

Figure 11-1 Query Evaluators: Nesting and Federation

Objects may participate in the Query Service in two ways. The simplest involves any CORBA object as is. The Query Evaluator is then responsible for evaluating the query predicate and performing all query operations by invoking operations on that object through its published OMG IDL interfaces. Any non-supported operations trigger exceptions. This mechanism provides the greatest generality, including support for all CORBA objects, but with the least optimization.

In a more involved manner, objects participate as members of a collection, either explicit or implicit. The collection supports a specific query interface (that is, the collection is itself a Query Evaluator). In this case, the Query Evaluator passes the query predicate to the collection, which then evaluates the predicate and performs query operations on an appropriate member object, receives any result, combines such results with all other participating object results, and returns this to the caller. This accomplishes the nesting, by passing the query evaluation on to a lower level. Such nesting may continue to an arbitrary number of levels, without limit.

This second way allows Query Evaluators or any associated native query systems to evaluate the query using the internal optimization at their disposal. This is expected to include faster access, caching, and indexing. Interpretation of names embedded in query predicates is determined by the Query Evaluator or its associated native query systems.

The Query Service specification does not define evaluation, indexing or optimization mechanisms. These are in the province of the implementor and may vary significantly in different environments. The Query Service simply provides a mechanism for passing the query to such systems and allowing their optimizations to take effect.

Collections

The Query Service provides definitions and interfaces for creating and manipulating collections of objects. These (explicit) collections may form both the scope to which a query may be applied and the result of the query, when the result is one or more objects.

The collections are defined as objects, with methods for adding and removing members. They may be arbitrary in nature. In particular, they are not limited to type extents, as in some object systems, though type extents are examples of such collections. They may map directly to collections managed by native query systems, for optimization, and may also include arbitrary CORBA objects.

Associated iterators are defined to allow manipulation of collections, including traversal over and retrieval of the objects within the collections. Such iterators allow a constant interface that can be invoked and implemented for arbitrary situations, including mixtures of general CORBA objects; native query system collections; highly distributed collections that could not be simultaneously accessed; collections across multiple heterogeneous products and systems; very small collections; and very large collections that could not be materialized physically.

Queryable Collections for Scope and Result

For collections to serve as both the result of a query and as a scope for another query, these collections must themselves be Query Evaluators. Such collections are called Queryable Collections. They support both the Query Evaluator and collection interfaces, as illustrated in Figure 11-2.

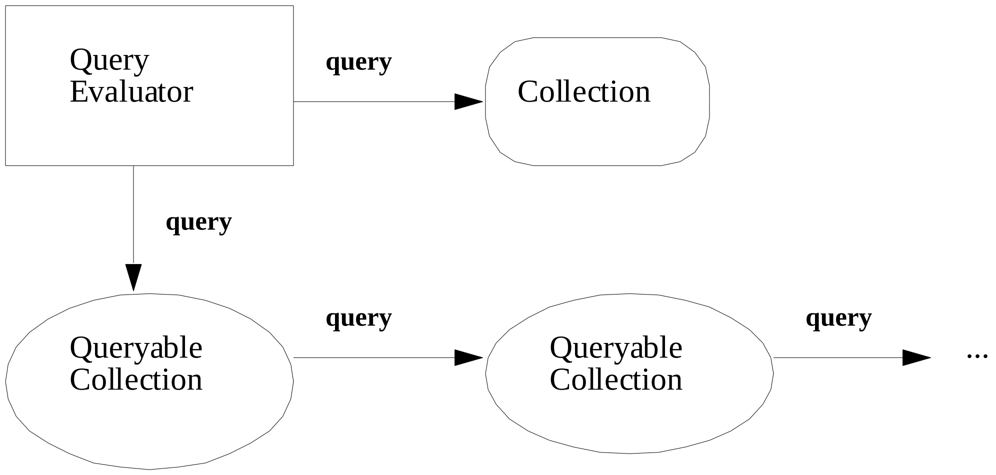

Figure 11-2 Queryable Collections

One of the issues that arises in using Queryable Collections is scoping in a nested environment. If the collection being queried allows adding arbitrary objects, and if objects are then added which are outside the scope of the evaluation mechanism of the Queryable Collection, then the Queryable Collection would have to provide the full functionality of a top-level Query Evaluator, evaluating predicates on arbitrary CORBA objects. This would defeat the purpose of nesting.

To solve this problem, we allow Queryable Collection implementations, in response to the invocation of the add and replace operations, to internally decide whether to add or replace the specified object, and to raise an exception if they decide not to. This allows arbitrary Queryable Collections—which are always supported at the top Query Evaluator level, and sublevel implementations that scope Queryable Collections to their own domain—to use whatever local mechanisms their (possibly pre-existing) query engines use. Examples of local mechanisms include optimization capabilities such as physical and logical indices; clustering; caching, and so forth.

Query Objects

Since queries can be complex and resource-demanding, there are numerous circumstances under which one would like to:

- Use graphical means to construct a query.

- Save a query and re-execute it later on, maybe with different set of search parameters.
- Precompile a query for later execution; this may be for the purpose of syntax and semantics checking and/or query optimization.
- Execute a query in an asynchronous manner; go do something else and come back for the result.
- Check the status of a long-running query and decide whether to continue or abort.

The Query Service provides the preceding capabilities and extensions through the use of Query objects. A Query object is created by calling a Query Manager, which is a more powerful form of Query Evaluator. Once created, a client of the Query object can:

- Use whatever means appropriate to construct the query specification.
- Prepare the query for later execution.
- Execute the query any number of times, with the same or different set of search parameters.
- Check the status of the query.
- Obtain the result of the query.

How the Query object does the preceding tasks is determined by the Query object and its associated Query Manager.

11.1.4 Query Languages

By using a very general model and by using predicates to deal with queries, the Query Service is designed to be independent of any specific query languages. Therefore, a particular Query Service implementation can be based on a variety of query languages and their associated query processors.

However, in order to provide query interoperability among the widest variety of query systems and to provide object-level query interoperability, a Query Service provider must support one of the following two query languages: SQL Query or OQL.

(Query capability is commonly implemented in database systems, hence there are many products, tools, trained users, and experiences based on these implementations. To leverage this, we base the query language specification on SQL Query and OQL.)

- **SQL Query.** Specifically, SQL-92 Query, which is defined in Chapter 7 (Entry SQL), and Sections 13.7, 13.8 and 13.10 (Entry SQL) of Reference 1 on page 11-27. SQL Query is used as the generic term to denote the evolution of SQL-92 Query. That is, it is envisioned that SQL-92 Query will evolve into SQL-9x Query, and so forth. These will be future versions of SQL Query. SQL-92 Query is the current version.
- **OQL.** Specifically, OQL-93, which is defined in Chapter 4 of Reference 4 on page 11-27. OQL is used as the generic term to denote the evolution of OQL-93. That is, it is envisioned that OQL-93 will evolve into OQL-9x, and so on. These will be future versions of OQL. OQL-93 is the current version.

For those Query Service providers who intend to provide only basic object-level query interoperability (for example, to support the needs of the Life Cycle Service or Property Service), the following must also be supported:

- **OQL Basic**. Specifically, OQL-93 Basic, which is defined in Sections 4.11.1.2, 4.11.1.3, 4.11.1.4, 4.11.1.5, 4.11.1.6 (**set** only), 4.11.1.7 (except **first** and **last**) and 4.11.1.10 in Reference 4 on page 11-27.

Ideally we would like to specify a single query language, for complete query interoperability. The most widely used query language in currently available query systems is SQL-92 Query, which does not support full object query capabilities. OQL-93 does support full object query capabilities and contains a near- (but not exact) subset of SQL-92 Query. Including SQL-92 Query provides the widest interoperability with the most query systems, while including OQL-93 provides full OMG Object Model support and full object query capabilities.

X3H2 and ODMG have started working together toward merging SQL Query and OQL with the goal of specifying a single standard query language. As SQL Query and OQL evolve, the OMG will revise of the Query Service to conform to future changes.

SQL Query

In the relational database world the accepted standard for database language is SQL-92 (Reference 1 on page 11-27). The ANSI X3H2 committee is working on a new version, SQL3 (Reference 5 on page 11-27), which will include object extensions, among other things. The committee is still working on the details of the modeling constructs; the object model under consideration is different from the OMG's Object Model. It is important for the eventual SQL object model to be fully compatible with the OMG Object Model so that SQL Query, the query subset of SQL, can serve as the query *lingua franca* in the OMG environment.

SQL-92 is a full database language. Functionally, it consists of the following types of language statements: schema; data; transaction; commection; session; dynamic; diagnostics; and embedded exception declaration. Among these, only a subset of data statements deal directly with query. This subset is defined to be SQL-92 Query. SQL-92 Query basically deals with query over tables (special kind of collections) of rows (special kind of dynamic data structures). As such, it concerns with a sub-domain of object query.

OQL

In the object database world the leading standard is ODMG-93 (Reference 4 on page 11-27). The ODMG-93 standard includes an object model, based on the OMG's Core Object Model, with extensions, to form the proposed object database profile. Also included is the Object Definition Language, ODL, which is a strict superset of IDL, providing a means to define objects in this profile model. All extensions, including attributes and relationships, are visible in the object interfaces as operations, and hence remain compatible with OMG IDL and the OMG architecture.

ODMG-93 also includes OQL (that is, OQL-93). OQL-93 is an adaptation of the SQL-92 Query capability to extend to all objects in the ODMG object model. It includes the ability to include operation invocation in queries, to query over object inheritance hierarchies, to invoke inter-object relationships, and to query over arbitrary collections. OQL-93 is a query-only language; that is, it allows evaluation of a predicate and a returned result, but includes no specific constructs for object modification. The ability within OQL-93 to invoke operations provides the insert, update and delete capability without violating encapsulation.

The OQL-93 syntax and semantics are not exactly compatible with SQL-92 Query. However, ODMG is working with X2H2 to address this issue. It is important for the eventual OQL to be fully compatible with SQL Query so that there is only one standard query language. .

SQL Query = OQL

Both X3H2 and ODMG have agreed upon a vision of the evolution of SQL Query and OQL, as illustrated in Figure 11-3.

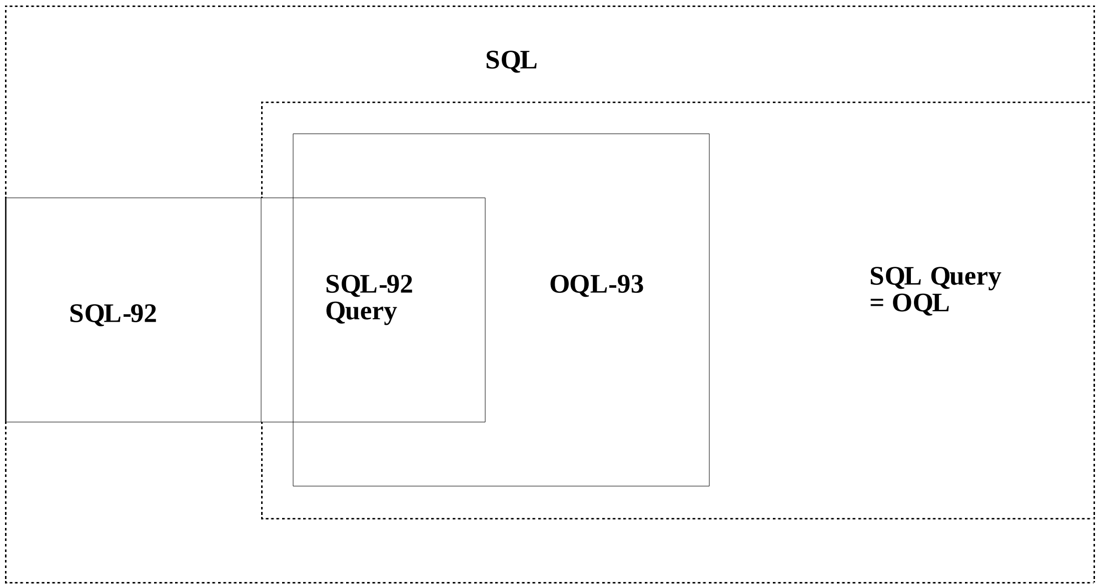

Figure 11-3 SQL Query = OQL

In Figure 11-3, solid lines indicate existing, defined specifications, while dotted lines indicate future specifications. As can be seen, SQL-92 Query is the query portion of SQL-92. OQL-93, being a query only language and having object features, overlaps with SQL-92 and is almost exactly compatible with it.

SQL-92 will evolve toward a future SQL, which is a full database language. OQL-93 will evolve toward a future OQL. The agreement from X3H2 and ODMG is to make the query subset of SQL, SQL Query, and OQL identical so that there is a single, common query language specification.

11.1.5 Key Features

The following are key features of the Query Service:

- Provides operations of selection, insertion, updating, and deletion on collections of objects. The objects may be transient or persistent, local or remote; the objects may have arbitrary attributes and operations.
- Accommodates different granularity of objects accessed by queries, including good support for high performance access to fine-grained objects.
- Allows the scope of the objects accessible in and via the collections that are the immediate operands of the query operations.
- Supports querying and/or returning complex data structures.
- Supports operating on user defined collections of objects.
- Supports operating on other kinds of collections and sets.
- Allows the use of attributes, inheritance, and procedurally-specified operations in the query predicate and in the computation of results.
- Allows the use of available interfaces defined by OMG-adopted specifications.
- Allows the use of relationships for navigation, including testing for the existence of a relationship between objects.
- Does not require breaking the encapsulation provided by the interfaces to objects.

In addition, the Query Service:

- Provides an extensible framework for dealing with object query.
- Is independent of the specific syntax and semantics of the query language used. The query language can be SQL Query, OQL, a graphical query language, or any other suitable object query language. In order to provide query interoperability among the widest variety of query systems and object-level query interoperability, a Query Service provider must support either SQL Query or OQL (OQL Basic with basic object-level interoperability) as specified in Section 11.1.4 on page 11-6.
- Allows for associative query and navigational query.

11.2 Service Structure

11.2.1 Overview

The Query Service defines two types of service. The specification is organized around these types.

Type One: Collections

The *Collection* and *Iterator* interfaces define the interfaces to create and manipulate collections of objects. The *Collection* interface is defined with operations for adding, retrieving, replacing, and removing member objects. The collections that it represents may be arbitrary in nature. The *Iterator* interface is defined with operations for traversing over and retrieving objects within a collection.

Type Two: Query Framework

The Query Framework interfaces define a flexible and extensible framework for dealing with object query. The *QueryLanguageType* interface provides the scheme to use the OMG IDL type system to classify query language types. The *QueryEvaluator* interface defines the basic operation to evaluate a query. The result of the query, which can serve as the scope for further queries, is represented by the *QueryableCollection*. The *QueryManager* interface defines a more powerful QueryEvaluator which can be called upon to create arbitrary *Query* objects. Such objects can provide the capability for graphical query construction, pre-compilation and optimization, asynchronous query execution, and so forth.

11.2.2 Collection Interface Structure

The collection interfaces are arranged into the interface structure illustrated in Figure 11-4. Dotted arrows represent association.

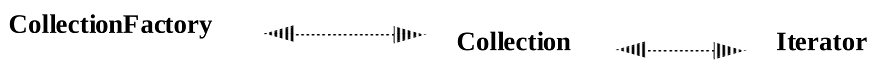

Figure 11-4 Collection interface structure

11.2.3 Query Framework Interface Hierarchy/Structure

The query framework interfaces are arranged into the interface hierarchy/structure illustrated in Figure 11-5. Solid arrows represent inheritance and dotted arrows represent association.

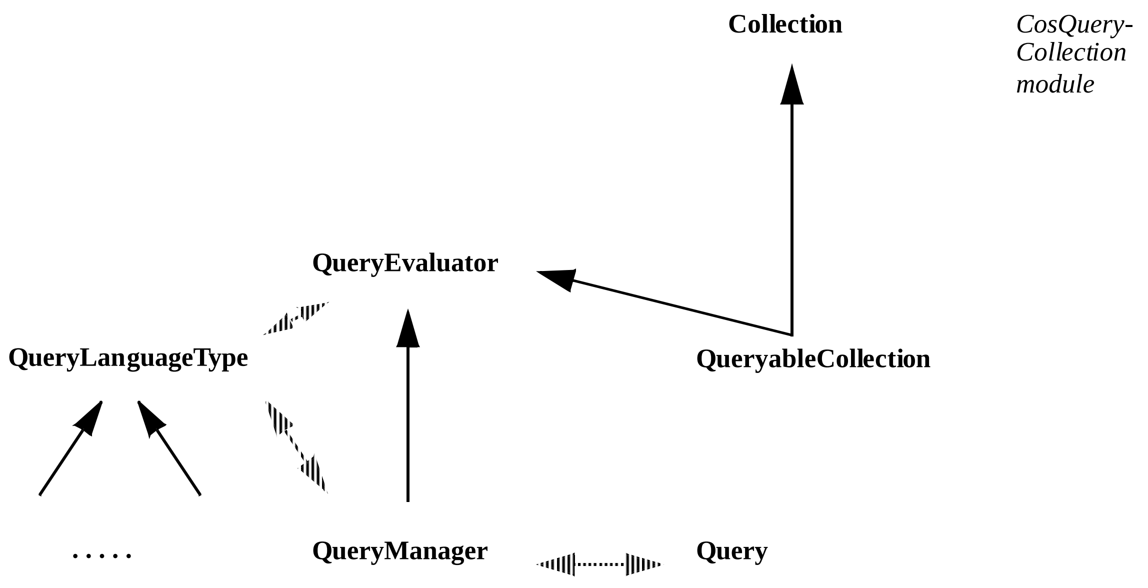

Figure 11-5 Query Framework interface hierarchy/structure

11.2.4 Interface Overview

The Query Service defines the interfaces to support the functionality described in Section 11.1 on page 11-1.

Table 11-1 and Table 11-2 give high level summaries of the Query Service interfaces. Collection interfaces are described in detail starting in the section Section 11.3 on page 11-12. Query interfaces are described in Section 11.5 on page 11-19.

Table 11-1 Interfaces defined in the *CosQueryCollection* module

Interface	**Purpose**
CollectionFactory	To create collections
Collection	To aggregate objects
Iterator	To iterate over collections

Table 11-2 Interfaces defined in the *CosQuery* module

Interface	**Purpose**
QueryLanguageType and its subtypes	To represent query language types
QueryEvaluator	To evaluate query predicates and execute query operations
QueryableCollection	To represent the scope and result of queries
QueryManager	To create query objects and perform query processing
Query	To represent queries

11.3 The Collection Model

11.3.1 Common Types of Collections

The Collection interface allows you to manipulate objects in a group. The objects that are part of a Collection are called its *elements*. Examples of common types of Collections are as follows:

- An *Equality Collection* has elements that can be checked for equality among each other. An example is a set.
- A *Key Collection* uses keys to identify elements (a key is part of an element). An example is a key bag.
- An *Ordered Collection* has its elements arranged so that there is always a first element, last element, next element, and previous element. Ordered Collections can be further classified as one of the following types:
  - A *Sequential Collection* has sequentially ordered elements. An example is a sequence.
  - A *Sorted Collection* has sorted elements. An example is a sorted set (which is also an equality Collection).

The Query Service defines only a top-level, basic Collection interface that supports query on arbitrary collections without restriction to any particular type. Subtyping can be used to map this basic Collection interface into a variety of collection classes, including the ANSI C++ Standard Template Library (STL), ODMGs, and others. The OMG Collection Service, available in the future, is expected to fit in similarly well.

11.3.2 Iterators

An Iterator is a movable pointer into a Collection. An Iterator is created in association with a Collection and can be used by a client to move through the member elements of the Collection. When an Iterator is created for an ordered Collection, it points to the beginning or the first element of the Collection. A series of next operations move it through subsequent elements until it passes through the last element and points to the end of the Collection. For unordered Collections, the elements are visited in an arbitrary order. Each element is visited exactly once.

The Iterator interface allows traversing a Collection in a way that works consistently for arbitrarily large Collections. In addition to the next operation, which can be used to move through the next element, it provides a reset operation to restart the iteration. Multiple Iterators can be created to maintain state concerning traversal of the same or different Collections.

The behavior of an Iterator can become undefined if elements are added to or deleted from its associated Collection. This means that its behavior depends upon the type and implementation of the Collection. In particular, an Iterator may become invalid as a result of such actions. Once an Iterator becomes invalid, it must be reset before it can be used for traversal again.

11.4 The CosQueryCollection Module

The *CosQueryCollection* module defines the Collection interfaces of the Query Service. In particular, it defines the

- *CollectionFactory* interfaces, to create Collections.
- *Collection* interface, to represent generic collections.
- *Iterator* interface, to enumerate the Collections.

The *CosQueryCollection* module is shown below.

```
module CosQueryCollection {

    exception ElementInvalid {};
    exception IteratorInvalid {};
    exception PositionInvalid {};

    enum ValueType {TypeBoolean, TypeChar, TypeOctet, TypeShort,
TypeUShort, TypeLong, TypeULong, TypeFloat, TypeDouble,
TypeString, TypeObject, TypeAny, TypeSmallInt, TypeInteger,
TypeReal, TypeDoublePrecision, TypeCharacter, TypeDecimal,
TypeNumeric};
    struct Decimal {long precision; long scale; sequence<octet>
value;}
    union Value switch(ValueType) {
        case TypeBoolean: boolean b;
        case TypeChar: char c;
        case TypeOctet: octet o;
        case TypeShort : short s;
        case TypeUShort : unsigned short us;
        case TypeLong : long l;
        case TypeULong : unsigned long ul;
        case TypeFloat : float f;
        case TypeDouble : double d;
        case TypeString : string str;
        case TypeObject : Object obj;
        case TypeAny : any a;
        case TypeSmallInt : short si;
        case TypeInteger : long i;
        case TypeReal : float r;
        case TypeDoublePrecision : double dp;
        case TypeCharacter : string ch;
        case TypeDecimal : Decimal dec;
        case TypeNumeric : Decimal n;
    };
    typedef boolean Null;
    union FieldValue switch(Null) {
        case false : Value v;
    };
    typedef sequence<FieldValue> Record;

    typedef string Istring;
    struct NVPair {Istring name; any value;};
    typedef sequence<NVPair> ParameterList;
```

Figure 11-6 CosQueryCollection Module

```
module CosQueryCollection {

    interface Collection;
    interface Iterator;

    interface CollectionFactory {
        Collection create (in ParameterList params);
    };

    interface Collection {
        readonly attribute long cardinality;

        void add_element (in any element) raises(ElementInvalid);
        void add_all_elements (in Collection elements)
raises(ElementInvalid);

        void insert_element_at (in any element, in Iterator where)
raises(IteratorInvalid, ElementInvalid);

        void replace_element_at (in any element, in Iterator
where) raises(IteratorInvalid, PositionInvalid, ElementInvalid);

        void remove_element_at (in Iterator where)
raises(IteratorInvalid, PositionInvalid);
        void remove_all_elements ();

        any retrieve_element_at (in Iterator where)
raises(IteratorInvalid, PositionInvalid);

        Iterator create_iterator ();
    };


    interface Iterator {
        any next () raises(IteratorInvalid, PositionInvalid);

        void reset ();
        boolean more ();
    };
};
```

Figure 11-6 CosQueryCollection Module

11.4.1 The CollectionFactory Interface

The *CollectionFactory* interface defines an operation for creating an instance of a Collection.

Creating a Collection

```
Collection create (in ParameterList params);
```

This operation creates a new instance of a Collection. The factory is passed a list of parameters, one of which must be:

“initial_size”, type long

which represents an initial, estimated number of elements. The Collection is initially empty and may grow dynamically, both in elements and size. Other parameters that may be passed include, for example, “hints” relating to indexing, and so forth.

The `ParameterList` is defined to be a sequence of name-value pairs, of which the name is defined to be of type `Istring.` As is the case in the Naming Service, `Istring` is a placeholder for a future OMG IDL internationalized string data type.

11.4.2 The Collection Interface

The *Collection* interface defines operations to:

- Add elements
- Replace elements
- Remove elements
- Retrieve elements

to and from a collection and an operation to create iterators for traversing the collection.

The element type of a collection can be any. This is designed to accommodate generality. For most common queries, the result collections tend to consist of elements that are records or objects. For some specific queries, however, the result collections may consist of elements of any data type.

`Record` is defined to be a sequence of `FieldValues`. A `FieldValue` may be Null or may have a value. This is designed to provide direct mapping to similar features available in a wide variety of existing query systems. The type of a `FieldValue` can be one of the OMG IDL base types, string, Object or one of the suggested mappings to SQL data types: TypeSmallInt; TypeInteger; TypeReal; TypeDoublePrecision; TypeCharacter; TypeDecimal; and TypeNumeric. (TypeFloat is the same as that defined for the OMG IDL base type.)

Determining the Cardinality

```
readonly attribute long cardinality;
```

This attribute identifies the number of elements that a Collection contains.

Adding an Element

```
void add_element (in any element) raises(ElementInvalid);
```

This operation adds an element to a Collection. Behaviors of all Iterators of the Collection become undefined when the element is added.

A Collection implementation, in response to the invocation of the add_element() operation, may internally decide whether to add the specified element, raising the `ElementInvalid` exception if it decides not to add it. As discussed in “Queryable Collections for Scope and Result” on page 11-5, this allows sublevel Query Evaluator implementations that scope Queryable Collections to their own domain.

Adding Elements from a Collection

```
void add_all_elements (in Collection elements) raises
(ElementInvalid);
```

This operation adds all elements of the input Collection to a Collection. The elements are added in the Iterator order of the input Collection and are consistent with the semantics of add_element(). This operation is really a sequence of add_element(). If any elements are added, behaviors of all Iterators of the Collection become undefined.

Inserting an Element

```
void insert_element_at (in any element, in Iterator where)
raises(IteratorInvalid, ElementInvalid);
```

This operation inserts an element to a Collection at the position pointed to by the input Iterator. Behaviors of all Iterators of the Collection, except the input Iterator, become undefined when the element is inserted.

If the input Iterator is invalid, the `IteratorInvalid` exception will be raised. The `ElementInvalid` exception will be raised as it is for the add_element() operation.

Replacing an Element

```
void replace_element_at (in any element, in Iterator where)
raises(IteratorInvalid, PositionInvalid, ElementInvalid);
```

This operation replaces the element of a Collection, pointed to by the input Iterator, with the input element. The input element must have the same positioning property as the replaced element. (Only equality Collections and key Collections have positioning property.)

If the input Iterator is invalid, the `IteratorInvalid` exception will be raised. If the Iterator does not point at an element, the `PositionInvalid` exception will be raised. The `ElementInvalid` exception will be raised in the same manner as it is for the add_element() operation.

Removing an Element

```
void remove_element_at (in Iterator where) raises
(IteratorInvalid, PositionInvalid);
```

This operation removes the element of a Collection, pointed to by the input Iterator. After removal, behaviors of all Iterators of the Collection become undefined.

If the input Iterator is invalid, the `IteratorInvalid` exception will be raised. If the Iterator does not point at an element, the `PositionInvalid` exception will be raised.

Removing all Elements

```
void remove_all_elements ();
```

This operation removes all elements from a Collection. After removal, behaviors of all Iterators of the Collection become undefined.

Retrieving an Element

```
any retrieve_element_at (in Iterator where) raises
(IteratorInvalid, PositionInvalid);
```

This operation retrieves the element of a Collection, pointed to by the input Iterator.

If the input Iterator is invalid, the `IteratorInvalid` exception will be raised. If the Iterator does not point at an element, the `PositionInvalid` exception will be raised.

Creating an Iterator

```
Iterator create_iterator ();
```

This operation creates an Iterator for a Collection. The Iterator is initially set at the beginning of the Collection.

11.4.3 The Iterator Interface

The Iterator interface defines operations to:

- Access and navigate through elements of a collection
- Reset the iteration
- Test for completion of an iteration

Accessing the Current Element

```
any next () raises(IteratorInvalid, PositionInvalid);
```

This operation retrieves the element of a Collection, pointed to by the Iterator, and advances the Iterator position. The operation will raise the `IteratorInvalid` exception if the Iterator is invalid, and the `PositionInvalid` exception if the Iterator does not point at an element.

Resetting the Iteration

```
void reset ();
```

This operation resets the iteration to begin anew. The position of the Iterator is reset to the beginning of a Collection.

Testing for Completion of an Iteration

```
boolean more ();
```

This operation returns *true* if there are more elements to be accessed and *false* if there are not.

11.5 The Query Framework Model

The Query Framework interfaces provide an extensible framework for dealing with query. This is accomplished in two ways. First, by providing a standard, generic set of object interfaces for handling query. Second, by providing extensibility so that these object interfaces can be subtyped for further functionality.

The Query Framework interfaces define two levels of interfaces. The base level consists of *QueryEvaluator* and *QueryableCollection* interfaces and provides the minimal functionality for query. The advanced level consists of *QueryManager* and *Query* interfaces and provides an extensible functionality for dealing with all aspects of query.

11.5.1 Query Evaluators

A Query Evaluator is any object that supports the operation to evaluate a query. It can be a single object, an implicit collection of objects, or an explicit collection of objects (particularly a Queryable Collection, as discussed in Section 11.5.2 on page 11-20). An example of a Query Evaluator that manages implicit collections of persistent objects is a database system.

The result of a query evaluation can be anything. In most cases, it is a Queryable Collection, as illustrated in Figure 11-7. (The solid arrow represents operation invocation and the dotted arrows represent association.)

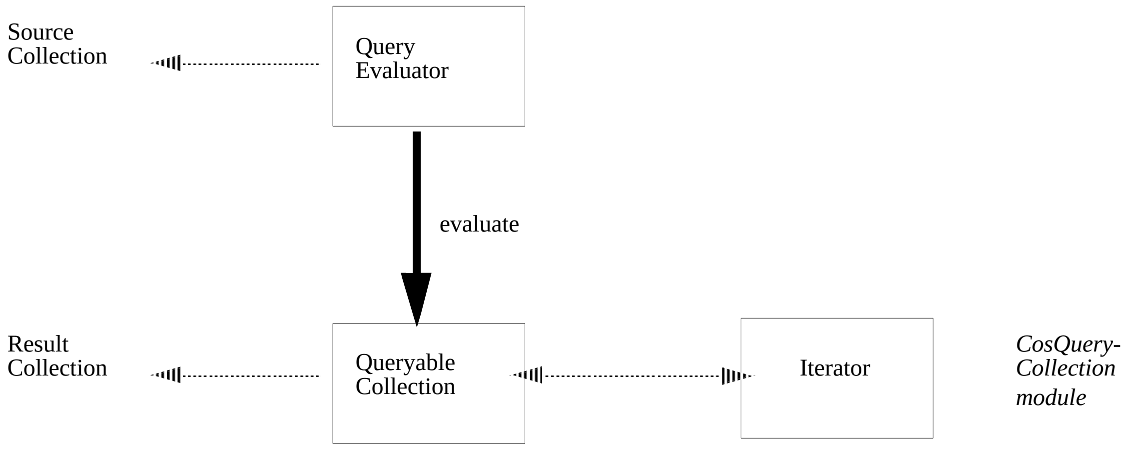

Figure 11-7 Query Evaluator and Queryable Collection

11.5.2 Queryable Collections

A Queryable Collection supports the *QueryEvaluator* interface and, therefore, can be used not only to represent the result of a query that consists of one or more objects, but also to define the scope to which further queries may be applied. An especially interesting kind of Queryable Collection is the type extent, whose member objects are instances of a certain object type.

A Queryable Collection evaluates a query by either invoking the evaluation operations on its member objects if they are Query Evaluators—or by evaluating the query predicate on the attributes and operations of its member objects if they are not—and by combining the results from such invocations and evaluations. As such, the query predicate must be a valid predicate for the Queryable Collection object and its member objects. If any one of its member objects is a Queryable Collection, the predicate (the applicable part, that is) must further be a valid predicate for its member objects, and so on. Therefore, the *QueryableCollection* interface provides a mechanism for nesting queries to an arbitrary number of levels.

11.5.3 Query Managers

A Query Manager is a more powerful form of Query Evaluator. It provides the operation to create Query objects. Working in tandem with a Query object, it manages the overall query processing and monitors the query execution. The *QueryManager* contains the universe of collections of objects over which queries can be specified. A specific query, as represented by a Query object, operates on a subset of this universe of collections.

The relationship between a Query object and its Query Manager is shown in Figure 11-8. (Dotted boxes represent logical entities; dotted arrows represent logical associations.)

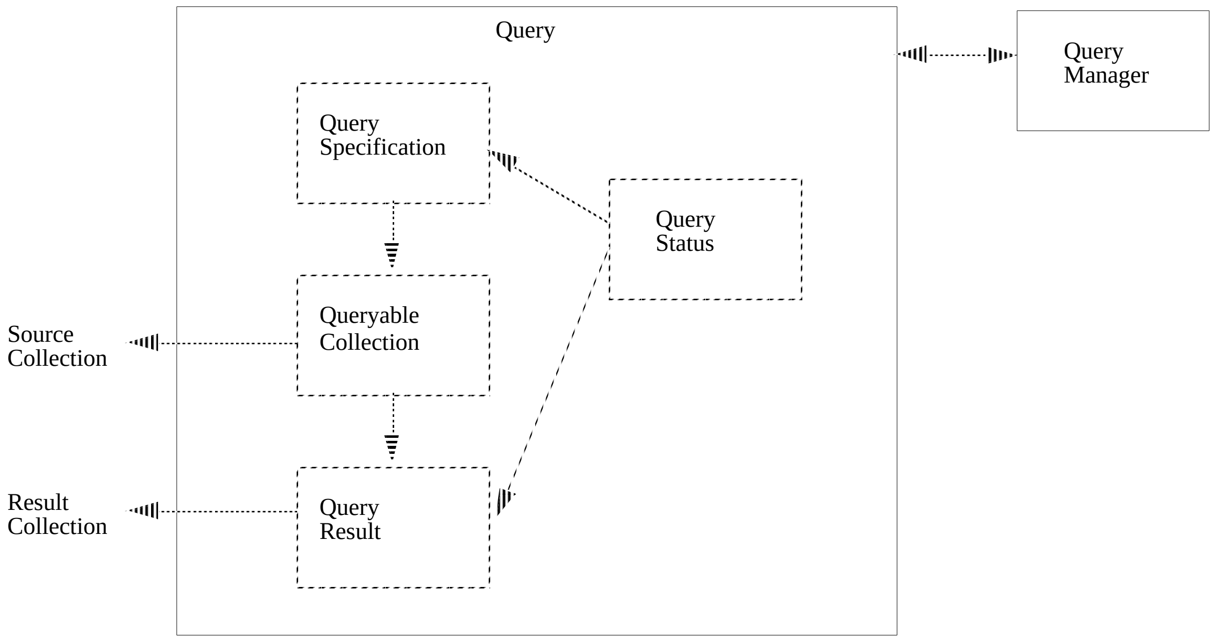

Figure 11-8 Query Manager and Query Object

11.5.4 Query Objects

A Query object represents a query and logically consists of the query specification, query status and query results. In addition, it contains the reference, either explicitly or implicitly through the Query Manager, to the queryable collection that defines its scope.

The Query object is responsible for composing and containing a query specification, including parameters. The query specification may be represented in the form of text, graphic, etc. A user may select a subset of the query specification to be executed in a query. This is particularly useful for query debugging. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for composing and selecting the query specification.

The Query object is responsible for maintaining the status information and log information regarding a query. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for displaying the status information.

The Query object also contains the results of a query. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for browsing query results. For example, successive results may be appended to previous results or replace them. A user may browse query results by specifying the version numbers, and so forth.

11.6 The CosQuery Module

The *CosQuery* module defines the query framework interfaces of the Query Service. In particular, it defines the following interfaces:

- *QueryLanguageType* interfaces to denote query language types.
- *QueryEvaluator* interface to represent query evaluators.
- *QueryableCollection* interface to denote collections which can serve as the result as well as the source of a query.
- *QueryManager* interface to create queries and perform query processing.
- *Query* interface to represent queries.

The *CosQuery* module is shown below.

```
 module CosQuery {

    exception QueryInvalid {string why};
    exception QueryProcessingError {string why};
    exception QueryTypeInvalid {};

    enum QueryStatus {complete, incomplete};

    typedef CosQueryCollection::ParameterList ParameterList;
    typedef CORBA::InterfaceDef QLType;

    interface QueryLanguageType {};
    interface SQLQuery : QueryLanguageType {};
    interface SQL_92Query : SQLQuery {};
    interface OQL : QueryLanguageType {};
    interface OQLBasic : OQL {};
    interface OQL_93 : OQL {};
    interface OQL_93Basic : OQL_93, OQLBasic {};

    interface QueryEvaluator {
       readonly attribute sequence<QLType> ql_types;
       readonly attribute QLType default_ql_type;

       any evaluate (in string query, in QLType ql_type, in
ParameterList params) raises(QueryTypeInvalid, QueryInvalid,
QueryProcessingError);
    };

    interface QueryableCollection : QueryEvaluator, CosQueryC-
ollection::Collection {};

    interface QueryManager : QueryEvaluator {
       Query create (in string query, in QLType ql_type, in
ParameterList params) raises(QueryTypeInvalid, QueryInvalid);
    };
```

```
   interface Query {
      readonly attribute QueryManager query_mgr;

      void prepare (in ParameterList params) raises(QueryPro-
cessingError);
      void execute (in ParameterList params) raises(QueryPro-
cessingError);

      QueryStatus get_status ();
      any get_result ();
   };

};
```

11.6.1 The QueryLanguageType Interfaces

The *QueryLanguageType* interfaces consist of seven interfaces that form the interface hierarchy illustrated in Figure 11-9.

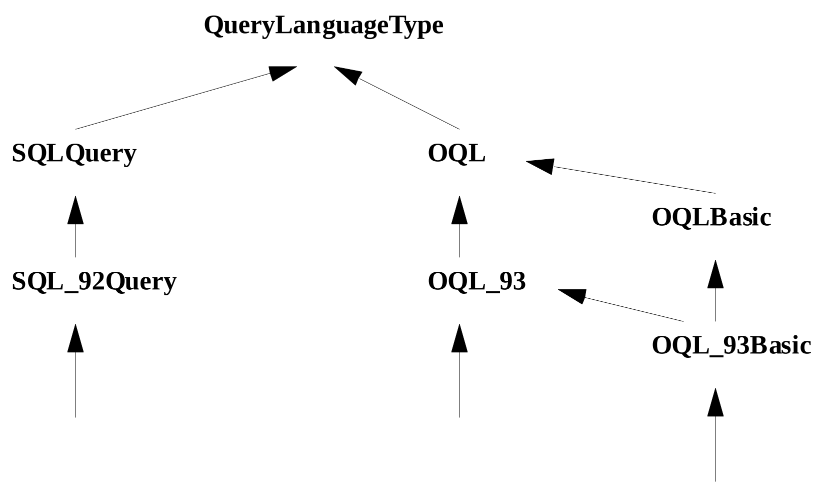

Figure 11-9 QueryLanguageType Interface Hierarchy

A Query Service provider is expected to use subtyping from SQL_92Query, OQL_93 or OQL_93Basic to denote the query language that it supports. For example, if a Query Service provider supports a query language, Object SQL, which complies with both SQL-92Query and OQL-93Basic, then its interface type, ObjectSQL, should be defined to be a subtype of SQL_92Query and OQL_93Basic:

```
interface ObjectSQL : SQL_92Query, OQL_93Basic {};
```

11.6.2 The QueryEvaluator Interface

The *QueryEvaluator* interface defines an operation for evaluating queries. It lets a client determine the query language types, and the default one, that it supports.

The result type of a query can be any. This is designed to accommodate generality. For most common queries, the results tend to be Collections (mostly of records or objects). For some specific queries, however, the result may be of any data type.

Determining the Supported Query Language Types

```
readonly attribute sequence<QLType> ql_types;
```

This attribute identifies the query language types supported by the *QueryEvaluator*.

Determining the Default Query Language Type

```
readonly attribute QLType default_ql_type;
```

This attribute identifies the default query language type supported by the *QueryEvaluator*.

Evaluating a Query

```
any evaluate (in string query, in QlType ql_type, in
ParameterList params) raises(QueryTypeInvalid,
QueryInvalid, QueryProcessingError);
```

This operation evaluates a query and performs required query processing. If the query language type is not specified, the default query language type is assumed.

The query language type specified must be supported by the QueryEvaluator. Otherwise, the `QueryTypeInvalid` exception is raised. If the query syntax or semantics is incorrect or if the input parameter list is incorrect, the `QueryInvalid` exception is raised. If any error is encountered during query processing, the `QueryProcessingError` exception is raised.

11.6.3 The QueryableCollection Interface

The QueryableCollection interface is a subtype of both the *QueryEvaluator* and *CosQueryCollection::Collection* interfaces. Any collection that supports this interface can be used to represent the result of a query that consists of one or more objects. It can also be used to define the scope to which further queries may be applied.

11.6.4 The QueryManager Interface

The *QueryManager* interface is a subtype of the *QueryEvaluator* interface. It defines an additional operation for creating Query objects. The QueryManager interface works in tandem with a Query object in managing the overall query processing and monitoring the query execution.

Creating a Query Object

```
Query create (in string query, in QlType ql_type, in
ParameterList params) raises(QueryTypeInvalid,
QueryInvalid);
```

This operation creates a Query object representing the input query. If the query language type is not specified, the default query language type is assumed.

The query language type specified must be supported by the QueryManager. Otherwise, the `QueryTypeInvalid` exception is raised. If the query syntax or semantics is incorrect or if the input parameter list is incorrect, the `QueryInvalid` exception is raised.

11.6.5 The Query Interface

The *Query* interface defines operations to:

- Prepare the query for execution
- Execute the query
- Determine the preparation and execution status of the query
- Obtain the result of the query

Determining the Associated Query Manager

```
    readonly attribute QueryManager query_mgr;
```

This attribute identifies the *QueryManager* associated with the Query object.

Preparing the Query for Execution

```
void prepare (in ParameterList params) raises
(QueryProcessingError);
```

This operation performs the necessary processing, including optimization, on the query so that it is ready for execution. Query preparation may be carried out in cooperation with the associated *QueryManager.*

If the input parameter list is incorrect or if any error is encountered during query preparation, the `QueryProcessingError` exception is raised.

Executing the Query

```
void execute (in ParameterList params) raises
(QueryProcessingError);
```

This operation executes the query. If the query has not been prepared before, it will prepare the query first. Query execution may be carried out in cooperation with the associated *QueryManager*.

If the input parameter list is incorrect or if any error is encountered during query execution, the `QueryProcessingError` exception is raised.

Determining the Query Status

```
QueryStatus get_status ();
```

This operation returns the preparation and/or execution status of the query. This may be carried out in cooperation with the associated *QueryManager*.

Obtaining the Query Result

```
any get_result ();
```

This operation returns the result of the query.

11.7 References

1. American National Standard X3.135-1992, *Database Language - SQL*, January, 1993.
2. Object Management Group. CORBA: *Common Object Request Broker Architecture and Specification*. Published by the OMG, Framingham, MA. 1995.
3. Object Management Group. *Object Services RFP 4*, OMG Document Number 94.4.18, May, 1994.
4. Cattell, R.G.G. (ed), *The Object Database Standard: ODMG-93, v1.2,* Morgan Kaufmann Publishers, San Mateo, California. 1994.
5. Melton, Jim (ed), *SQL3 Part 2: Foundation*, ANSI X3H2-94-329, August, 1994.