CONSTRUCTING MULTIDATABASE COLLECTIONS USING AN EXTENDED ODMG OBJECT MODEL[3]

Adrian Skehill Mark Roantree
Interoperable Systems Group
Dublin City University
Dublin 9, Ireland.

ABSTRACT

Collections are an important feature in database systems. They provide us with the ability to group objects of interest together, and then to manipulate them in the required fashion. The OASIS project is focused on the construction a multidatabase prototype which uses the ODMG model and a canonical model. As part of this work we have extended the base model to provide a more powerful collection mechanism, and to permit the construction of a federated collection, a collection of heterogenous objects taken from distributed data sources

INTRODUCTION

The OASIS Project (ODMG Architectures for Specification of Interoperable Systems) is focused on the construction of a multidatabase prototype for usage in a healthcare environment (Roantree 1998). Research is conducted at Dublin City University with assistance provided by Iona Technologies with their Common Object Request Broker Architecture (CORBA) products, Versant through the usage of their object-oriented database, and St James's Hospital through the use of their extensive healthcare applications as a suitable test environment for the planned prototype.

This paper describes one of the sub-projects within the OASIS project where the collection mechanism of the ODMG model was extended to provide three key features. Firstly, OASIS Dynamic Collections are automatically updated through our Collection Manager; secondly our collections are capable of containing heterogenous objects; and thirdly collections from different databases can be merged to form a federated collection. We will refer to OASIS Dynamic Collections as Dynamic Collections during the rest of the paper. The paper is structured as follows: the remainder of this section briefly describes ODMG collections and motivates our need to extend the base model; §2 looks at the design architecture for implementing dynamic federated collections; §3 provides information on constructing local and global collections; §4 describes the construction of dynamic collections; §5 describes the prototype; and finally §6 offers some conclusions.

Motivation

The ODMG Object Model provides a generic Collection class, which defines the basic properties for each collection type (Catell & Barry 1997). Specific collection types inherit from this object providing more specific collections. The collections supplied by the ODMG Object Model are listed in *Table 1.*

Type	Description
set	Unordered collection of elements, no duplicates allowed.
Bag	Unordered collection of elements, which may contain duplicates
list	Ordered collection of elements.
array	Dynamically sized ordered collection of elements.
dictionary	Unordered sequence of key value pairs with no duplicate keys.

Table 1: *ODMG Collection Types*

The elements of a collection can be of any type, primitive literals, structures and abstract data types, objects and references to objects. However, a collection can hold one type, and all elements in the collection must be of that type.

In order to extract information from the collection, an iterator is used. This is a mechanism for accessing elements by traversing the collections. Iterators are defined in the Standard Template Library (Musser et al 1996) specification. This defines standards for collections, and how to extract information from them using iterators. There are two types of iterator defined by the ODMG standard, a generic iterator which moves forward through the elements of a collection, and a bidirectional iterator which allows bidirectional traversals of a collection.

[3] This work has been funded by Forbairt Strategic Research Grant ST/98/014.

The ODMG collection handling mechanism is static. A collection is defined, and must be initialised and maintained by the programmer. The contents of a collection may become invalid for any number of reasons, e.g. objects can be deleted, modified or new objects can be added. Our requirement is to export object collections from various hospital information systems to provide the global user with federated collections of objects of interest. Although no facility exists to perform manipulations on the data inside federated collections, an import facility can download the federated collection into a local database as snapshot data where OQL can be used to execute meaningful queries.

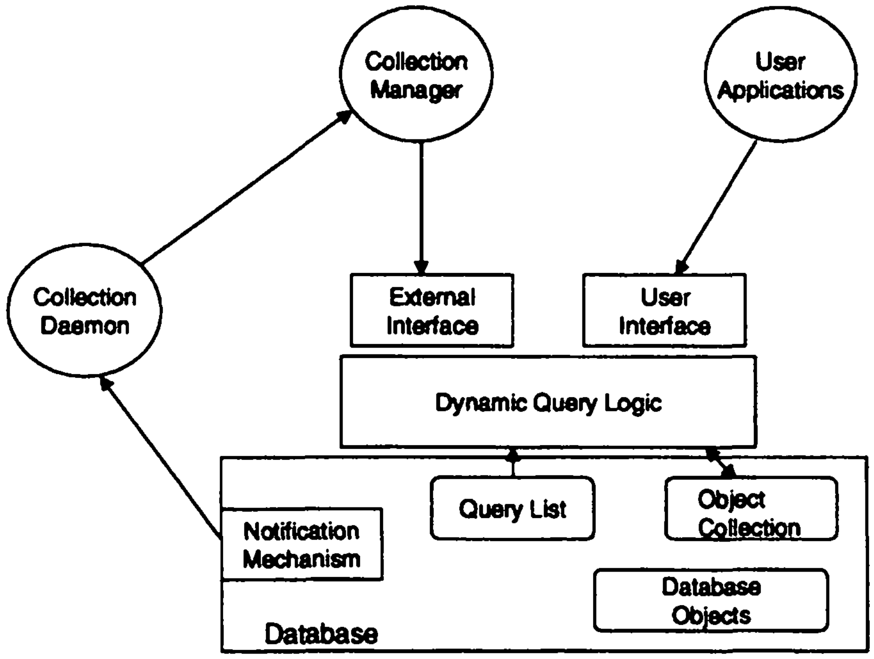

Figure 1: Architecture for *OASIS Dynamic Collections*

OASIS Dynamic Collections solve the problem of static collections by allowing the application developer to associate a query with each collection, and then forcing the database to maintain the contents of the collection based on this query. ***Figure 1*** demonstrates the architecture for Dynamic Collections. Each component has a specific task which will be explained over the course of this paper. A Dynamic Collection can contain different types of objects whereas ODMG collections can hold only one type of object. A number of queries can be used to define a collection, and their result is held internally in the collection. This allows a meaningful and diverse set of objects to be stored within a collection. The OASIS multidatabase kernel also allows Dynamic Collections to be merged, thus forming federated collections of data.

ARCHITECTURE FOR DYNAMIC COLLECTIONS

Maintenance Interface

The Maintenance Interface (MI) facilitates the specification of queries for collections. Multiple OQL queries can be specified for each collection. The MI also requires a name for each collection constructed.

- Naming the Collection. After creating an instance of a dynamic collection, it can be named, so that other applications will be able to easily retrieve a reference to this collection (and hence its contents) from the database. To name the collection, the method name_collection is used. This saves the name supplied in the database, and other applications will be able to query based on this name.
- Supplying the Query Definitions. A Dynamic Collection can have a number of queries defined in order to place heterogenous types inside the collection. After constructing the Dynamic Collection, any number of queries can be defined. A query is associated with a collection by calling add_query where the query is a string representing an OQL statement. If the OQL statement is either syntactically or semantically

incorrect, then an exception is thrown (in C++) and the query is refused.

- Initialising the Collection. Once all the queries have been added to the dynamic collection, it can then be initialised. This is a process where each of the queries associated with the collection are executed, and each object reference is added to the collection. It is achieved by **calling the** initialise method. Since this call actually triggers the initial creation of the contents of the collection, it may take some time as it has to execute each OQL query defined for the collection, and build a reference to the objects returned.

The following code segment, demonstrates how to use the maintenance routines to create, name, define and initialise a collection:

Example 1 *Maintenance Interface*

```
CDynCollection *col;
col = new (db_name, CDynCollection);
col->add_ query ("select Iname from Patients where pl5_ test>500");
col->name_ collection ("P15 Failures);
 col->initialise ();
```

In this example, a dynamic collection *cot is* created. The name of the collection is "P15 Failures", and the query that defines it is added using the add_query method. The initialize method is used to activate the Dynamic Collection, so that its contents will be updated as changes occur in the database.

Structural Interface

The Structural Interface (SI) provides information of the structure of the collection. It provides applications with details of the different types that are present in the collection. Calling the retrieve_object_names method returns a list of the names of classes that are stored in the collection. Information about each type supplied by the interface can be used to select objects of that type using the Data Interface. Browsers or other mechanisms that display the contents of the collection can use this information to format the display of data from the collection. The following code segment shows how the structural interface can be used. This is based on the code shown in the previous example.

Example *2 Structural Interface*

```
d_Set <d_Ref <d_String> > classes = col->retrieve_object_names ();
for (i = 0; i < classes .size(); i++)
{
cout << "Class  name is < " << classes[i] << "> ~ << endl;
   }
```

The class d_Set <d_String> is an array of strings, and is the storage for the list of types returned by the call to retrieve_object_names.

Data Interface

The Data Interface (DI) is used to extract objects (data) from the database. Access to data is through the get_obj_list method, which returns object references. The standard database locking mechanism is used to protect the list of objects retrieved. The application that processes the collection will have a lock on the objects to ensure that no other application can delete objects contained within the Dynamic Collection. If any objects are modified the change will be shown immediately after the commit point to the database. Once the lock has been removed from the list of objects returned, get_obj_list will have to be called again to refresh this list. The Dynamic Collection uses the standard ODMG collection mechanism to store its elements.

REFRESH MECHANISM

Once a Dynamic Collection has been constructed, the database engine is used to maintain the contents of the collection. A list of references to objects is materialized and stored so that the collection is not regenerated each time it is used by an application. However, this means that the collection must be refreshed each time an object associated with a collection is created, modified or deleted. Recalculating the entire collection each time could represent a large overhead for the database if there were a large number of objects involved. From a refresh point of view there are two issues here that have to be discussed: how the dynamic collection gets notified when a change has occurred, and how the collection refreshes itself in such a way as there is minimal impact on the performance of the database. We address both issues in the remainder of this section.

Notification Procedure

Applications which use an object oriented database generally retrieve objects from the database, and execute methods provided for the classes. Since Dynamic Collections are persistent classes containing object references, the database will not be able to refresh these collections. A refresh operation is quite complex as it will have to determine what objects to change in the collection (if any). An external component (the Collection Manager) was constructed to refresh Dynamic Collections, running continuously as a daemon program. Dynamic Collections provide an interface for the database to trigger a refresh. A method called **refresh** is invoked by the Collection Manager (CM) for this purpose, and is discussed in the next section. In order for the CM to refresh collections, a notification (or event) mechanism exists to provide details of change. The more refined this mechanism is, the more efficient the refresh mechanism will be. The ODMG Object Model does not contain a standard for event notification so proprietary event mechanisms must be built. The following features, embedded in the database engine attempt to optimise the refresh mechanism:

- The ability to define the events of interest only e.g. instance modification, creation and deletion, of a particular class.
- The ability to register the predicates of OQL statements which the dynamic collections are based on. Only events that effect these predicates will cause a notification message to be sent to the appropriate dynamic collection.
- The notification mechanism should supply the object identifier of an object that has been created, changed or deleted.

When the database using Dynamic Collections is started, a daemon process capable of defining and receiving events, the Collection Daemon (CD), must be started. The implementation of this application is dependent on the database containing the Dynamic Collections, as different databases have different event notification mechanisms. When the Collection Daemon starts, it queries the database for each Dynamic Collection that has been created in the database. For each collection found, the Collection Daemon will query the collection for

1. the predicates from each OQL statement defining the collection;

2. the class name corresponding to each predicate.

For each query which defines the dynamic collection, the collection daemon will retrieve the predicates for each OQL query. For example, in the following OQL query:

```
define extent Patients for patient;
select p from Patients p
where p.address.city = Cork
```

In this case the predicate is 'p.address.city = Cork'. So if a patient moves out of Cork, the collection will be notified that this object will need to be re-examined to confirm whether it should remain in the collection, or have its reference deleted. The Collection Daemon will also retrieve the names of classes corresponding to the predicate contained in the OQL statement. In the above example the class name will be *patient*. The Collection Daemon uses the event message received from database to determine which collection should be refreshed and uses the Collection Manager to refresh this collection only. Using this mechanism, we obtain the following advantages:

- The database engine is responsible for "watching" the objects of interest, which should be more efficient than having an application constantly checking the objects of interest in the database.
- Only the classes which are of interest to Dynamic Collections are being monitored. The predicate is a further refinement. The database will only notify the collection for objects satisfying the predicate.

The Collection Manager

The effort of the Collection Manager depends on the power of the database's event handling mechanism. With Versant (Versant 1997) it has a powerful event mechanism which is capable of returning the object reference for modified objects. The Collection Manager simply checks its collections to determine if the object reference is present, and re-evaluates for that object only.
Dynamic Collections maintain a state designed to provide information on the correctness of the list of objects it contains. *Table 2* provides a list and description of each of the five possible states.

State	Description
READY	The dynamic collection has been created but not initialised. The initialise method is invoked To ensure the collection has been activated. Activation of a Dynamic Collection means the Collection Manager can refresh its contents
EMPTY	The collection has been initialised, and no objects were returned as a result of the queries. The collection is still active so if any changes occur in the database that affect this collection, The collection will be updated.
BEING_REFRESHED	An event has occurred which has caused the collection to begin a refresh. No data will Be available from the collection while it is being refreshed.
VALID_COLLECTION	The collection is ready for queries. It has been successfully initialised and has a valid List of object references.
COL_ERROR	There is an error with the collection and no data is available from the collection. A typical Cause of this error is that the structure of the schema has changed, and one of the OQL Queries is longer valid.

Table 2: *Dynamic Collect: on States*

Handling Schema Changes

Another issue that Dynamic Collections must address is changes to the database schema. If a type structure is changed so that an OQL query will no longer execute (for example, the attribute that the query is based on is removed from the class definition), then the collection will no longer be able to refresh its contents based on the OQL query. Because of this, the user should check the state of the collection before attempting to access any elements in the collection. If the user attempts to retrieve elements from the collection and the state returned is COL_ERROR, then an exception is thrown which the application must address.

Since Dynamic Collections deal only with objects references they do not need to know the actual structure of the object types that are referenced by the collection. If new attributes are added to an object, this will not affect the operation of the collection. The collection can be updated using queries on this new attribute by calling the **add_query** method to add a new query to the collection.

CONSTRUCTING AND USING DYNAMIC COLLECTIONS

This section examines how dynamic collections can be used to **construct a 'cutdown'** federated schema and serve a useful purpose in a multidatabase prototype. Dynamic collections represent a convenient method to model export and federated schemes as they refresh themselves when changes occur.
Figure 2 illustrates how dynamic collections can be used to build a limited form of federated schema. Each database has had its local data model converted to the CDM of the multi database system. In this case the CDM is the ODMG Object Model. *Figure* 2 starts at the second layer of the traditional architecture of these systems (Sheth & Larson 1990) where the component schema resides. In order to construct export schemes it is necessary to query the schema repository of the component schema. Construction of a local dynamic collection (export schema) requires knowledge of the structure and meaning of the participating database. When constructing the federated dynamic collection the required information is provided by the Schema Server objects which provide distribution for local collections. Thus, a federated collection becomes a federation of exported collections of objects.

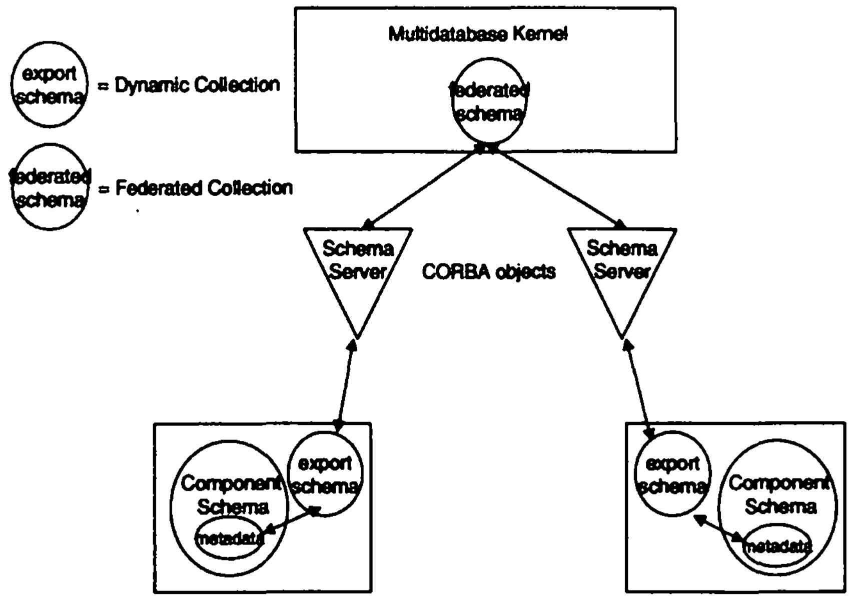

Figure 2: *Dynamic Collections in a Multidatabase Environment.*

Constructing the Export Schema

A component schema must be generated for each local database participating in the multidatabase system (Pitoura et al 1995, Sheth & Larson 1990). A Dynamic Collection can then be constructed to act as an *export* of data. As the component schema changes, so do the objects in the exported collections. This subsequently provides dynamic updates to federated schemes to ensure that schemes presented to the user is always correct. Metadata information is also present in Dynamic Collections so that a description of exported objects also exists.
Some process is required to export metadata to the multidatabase kernel and hold information on each database participating in the multidatabase system. This exists as an external application, illustrated as the Schema Server in *figure* 2. The Schema Server allows metadata to be queried and exported to the multidatabase kernel.

Accessing Federated Data

Using both the Structural Interface and the Data Interface, a constructing processor (Sheth & Larson 1990) is able to build a federated schema from any number of export collections. For any database participating in the multidatabase system, the Schema Server can access the local collections that can be used to construct the federated schema. For each export schema present in the database, the server imports the appropriate metadata into the multidatabase kernel with a unique id for the source of each metadata subset.
When the Schema Server has completed the importing process the MDB kernel will contain:

- metadata representing schemes from the participating databases;
- the various exported collections.

It is necessary for the multidatabase administrator to assist the constructing processor, as decisions will need to be made regarding name overlaps etc. The problems associated with schema integrations can be found in various research publications including (Batini et al 1986). At its most simple, it may be possible to define the federated schema as a collection of collections (i.e. export schemes). However, in most non-trivial situations, there will be some mediation required between the contents of the different export schemes that are to be integrated to form the federated collection.

Federated Collections

A federated collection consists of a collection of collections from local databases. A browser can easily be constructed to view data from multiple sources. Alternatively, the imported metadata can be used to create a

schema at the MDB kernel which can be used the store actual data, imported from local databases, and construct a multidatabase snapshot. Either application will need a bridge to allow the data to be retrieved from local databases. This bridge called the Data Server, exists as a CORBA object, and facilitates the supply of data from local sources.
For each collection present in the multidatabase collection, the data server will supply the contents of that collection to the calling application. The calling application will be responsible for decoding this information, and performing any additional processing before the data is displayed to the user.

APPLICATION PROTOTYPE

A working prototype has been built using Microsoft Visual C++ 5.0, Orbix 2.3c (1998), and Versant 5.0.8 (1997) and runs on a Windows NT platform. It comprises a number of components as follows:

- Collection Daemon (CD). This observes any events in the database that may affect the contents of any dynamic collection (C++/Versant).
- Collection Manager (CM). When notified by the collection daemon of a change, this application refreshes the dynamic collection(s) affected by the change (C++/Versant/Orbix).
- Data Server (DS). This application provides a data for application requests (C++/Orbix).
- Schema Server (SS).Generates the metadata to model the schema from each component database, and creates this information in the OASIS kernel database for applications to be able to query (C++/Orbix).

The last three components (CM, DS and SS) are CORBA servers, and must reside on an ORB which has access to the component database. Multiple implementations of these applications can exist, to allow for different platforms, database types etc. However, the interface will be common regardless of the implementation chosen.
Data was exported from legacy systems at St James' Hospital and imported directly into three component schema[4] which exist as Versant databases. Local collections were defined using our extended ODMG Collection Mechanism and subsequently merged to form federated collections. A browser was built to view federated collections. Test collection were specified by the hospital based on the imported data set.

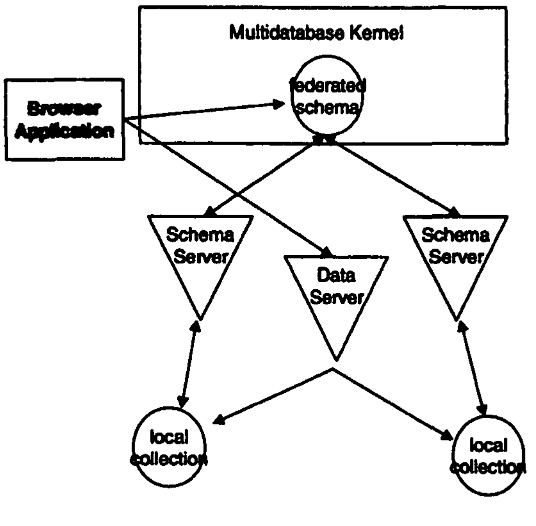

Figure 3: *Information Flow through the Prototype*

[4] A model converter was hard-wired specificaUy for this task. Future OASIS prototypes will facilitate both component schemes containing only mappings, and those containing imported snapshot data.

CONCLUSIONS

In this paper we described our extension to the ODMG object model to provide an enhanced collection handling mechanism. This research was motivated by the need to have collections automatically refreshed, and to provide federated collections of objects from distributed sources in a healthcare environment. Our work extends the ODMG model in three ways:

- Collections can contain heterogenous types;
- Collections are automatically refreshed when changes to the database require;
- Collections from separate databases can be merged to form a federated collection.

A prototype has been constructed and collections (supplied by St James Hospital) were used to test the system.

REFERENCES

Batini C., Lenzerini M., and Navathe S. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys, 3:(4),* 1986.

Catell R. and Barry D.(eds). *The Object Database Standard: ODMG* 2.0. Morgan Kaufinann, 1997.

Orbiz 2.S Documentation Suite. Iona Technologies, Dublin Ireland, 1998.

Landers D. Target Information Systems for the OASIS Prototype: A Description of the Systems and Solutions for Interoperability. *Oasis Report No. OAS-07,* Dublin City University, February 1999.

Musser D., Saini A. and Stepanov A. *STL Tutorial & Reference Guide: C++ Programming With the Standard Template Library* Addison Wesley, 1996.

E. Pitoura, O. Bukhres and A. Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys* 27:2, 1995.

Roantree M. The OASIS Multidatabase Architecture. *Oasis Report No. OAS-01,* Dublin City University, May 1998.

A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogenous and Autonomous Databases. *ACM Computing Surveys* 22:3, 1990.

Versant C++ Reference Manual 5.0. Versant Sofware, 1997.