An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and O_2.

Valery Soloviev*
The University of Toronto, Canada

Abstract

We present an analysis of three current object-oriented DBMS products: ONTOS, ObjectStore, and O2, as described by their available documentation. The most attractive feature of ONTOS and ObjectStore is their use of C++ as a user interface - a widespread object-oriented language.They also provide persistent data implementation, transaction and recovery mechanisms, and modern application development tool sets following the recommendations of [Atkinson et al. 89]. O2 was chosen for a well-developed data type system and end-user interface, and for its reputation from the literature.

1 Introduction

The field of object-oriented databases has rapidly evolved into a major area of database software research and development. A number of commercial OODBMS products have been introduced to the market in last two years. Recent books and papers present the current state of both research and commercial OODBMSs by comparing them with relational DBMSs and with each other. Very few discuss commercial products, however [Manola 89], [Bancilhon et al. 90b], [Kim et al. 89], [Zdonik et al. 90].

This paper presents an analysis of three current OODBMS products: ONTOS, ObjectStore, and O2, and is based on available user documentation ([O2 Technology 91a], [O2 Technology 91b], [O2 Technology 91c], [Object Design 90a], [Object Design 90b], [Ontologic 91]). The goal of analysis was to select one of the systems as a basic tool for software development. The systems were chosen for their support of C++ as a user interface (ONTOS, ObjectStore), or for a strong type system, powerful end-user interface tools like Look, and detailed description in the literature (O2).

*soloviev@cs.wisc.edu. Present address: Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton St, Madison, WI 53706

The rest of the paper is organized as follows. Section 2 discusses platforms and architectural features, Section 3 describes user interfaces and tool sets, Section 4 compares data models with example class descriptions, and Section 5 covers data aggregation. Query processing and concurrency control are described in Sections 6 and 7, Section 8 presents the ObjectStore cooperative group model, and finally Section 9 summarizes all the systems features in a table.

We do not deal with system performance because a lack of benchmark results covering all three systems would make such a comparison too superficial.

2 Platforms and architecture

ONTOS, Release 2.0, product of Ontologic Inc, supports UNIX, HP/Apollo, and OS/2 (in beta). **Object Store** [Lamb et al. 91], from Object Design Inc. was first available in 1990. The current Release 1.1 has two platforms: UNIX and Windows 3.0. The **O2 System** ([Deux et al. 91], [Bancilhon et al. 90a]) product of O2 Technology, France, was released in early 1991, and operates on Sun workstations. It will be ported to the HP-9000 and to MIPS processor-based workstations in the future. All three systems have a client/server architecture.

ObjectStore utilizes one server to support many client workstations. Each workstation can simultaneously access multiple databases on many servers, and is it also possible for a server to be resident on the same machine as a client.

An ObjectStore application requires 3 auxiliary processes:

- *Server.* The server handles all access to an ObjectStore file system, including a storage and retrieval of persistent data. A single application can use several databases, including databases on different file systems, which are handled by different servers.
- *Directory Manager.* ObjectStore organizes databases into *logical directories.* Two databases

in the same directory may be stored on different disks. The Directory Manager manages a hierarchy of directories, and maintains permission modes, creation dates, owners, and groups for each entry. Each site has one Directory Manager and one or more Servers.

- *Cache Manager.* Each node running one or more ObjectStore applications has a single Cache Manager, which manages each application's client cache of this node.

ObjectStore transfers referenced data by *segment* or by page across the network to an application cache. Transfer between an application's cache and it's virtual memory occurs on the page level.

In **ONTOS**, a *client* is created by linking the ONTOS Client Library into a C++ application. The *server* is a process which runs on every node, containing a portion of the database called an *area* . The *Binder* performs functions similar to the ObjectStore Directory Manager, but *cache* memory is managed by the server, i. e. there is a server cache instead of a client cache. A *segment* is a single unit of disk access and data transfer.

The **O2 System** has the *O2 Engine* as its core. It has three layers:

- The *Schema Manager* creates and processes classes, methods, and global names
- The *Object Manager* is responsible for object identities, and implements persistence and inheritance
- The Wisconsin Storage System (WiSS) [Chou et al. 85] is a storage system providing files, objects, and indices plus concurrency control and recovery services[Deux et al. 91a]

On the client, an application and the workstation version of WiSS form one unique process. The server can service one or more terminal application processes [Velez et al. 89]. The unit of transfer between the server and the client is a page, i.e. O2 implements a *page server* architecture [Deux et al. 91], as well as ObjectStore and ONTOS. The server does not need to be able to run user methods for this architecture. O2 used an *object server* model for a previous version [DeWitt et al. 90], but the overhead was too expensive.

3 User interfaces and development tools

ObjectStore offers a choice of a C library interface, a C++ library interface, or a DML preprocessor interface.

The C library interface and C++ library interface (implemented by the *cpp macros* facility) preserves the investment in existing C and C++ applications, and leverages current C or C++ compilers. However these interfaces give an access only to one of tools - Browser.

The DML preprocessor interface reduces the amount of code a user must write and offers advanced features such as parameterized types and query expressions. This development is provided a full set of tools: Schema Designer, Browser, and Debugger.

ObjectStore provides a low-cost migration path that allows users to easily:

1. convert a C program to a C++ program
2. convert C data stored in existing file systems to an ObjectStore database
3. integrate existing applications and libraries written in C with new ones written in C++

Converting a C application to use ObjectStore can be done easily. In order to make C data structures persistent 4 steps are required:

1. Declare the database

   ```
   database *current_db
   ```

2. Open the database and insert transaction boundaries

   ```
   current_db=
        database::open(''path/filename'');
   do_transactions (){
        <existing C application code>
   };
   ```

3. Change references to "malloc" to "new(db)" in the existing C application code
4. Delete the code that reads and writes C data structures from disk

ObjectStore manuals provide an example of an application program, where 134 lines of the C application code are transformed to 48 lines of "C with ObjectStore" application code after conversion by deletion of reads and writes. Only 6 lines of 48 are new or changed.

ObjectStore provides the following tools:

- Schema Designer: an interactive graphical design tool for developing, viewing, and evolving class schemas.

- Browser: a graphical tool for inspecting the contents of the database.

- Debugger: the extension of the GNU debugger.

ONTOS provides a C++ interface and the ONTOS SQL interface to application developers. The ONTOS *Classify* utility is a schema compiler. It reads C++ class definitions contained in C++ header files and creates a corresponding database schema. The schema can alternatively be created by the ONTOS DBDesigner tool. The ONTOS *cplus* utility is a preprocessor used to compile C++ files containing implementations of persistent classes. The code added by the utility performs initialization required for an object *activation* and procedure invocation. All source files containing implementations of *activation* constructors must be compiled using this utility.

ONTOS supports a hierarchy of *Type Definition Classes* , including schema definition classes describing each element of the class definition and access method iterators. The description of metaclasses allows one to define new classes and use them dynamically in extensible, interpretive applications. Instances of these classes, representing user-defined objects, are generated automatically by the Classify utility. Such classes are contained in ONTOS *Class Library* along with other ONTOS-provided classes, like Object, Iterator, Aggregate and others.

The ONTOS *SQL interface* provides SQL access to ONTOS interactively and via programs.

The ONTOS *DBDesigner* , a visual interactive schema designer and database browser, can be used to design, examine and modify the database schema and instances, and to create new object classes.

The **O2 System** supports two types of interfaces: the O2 environment - *O2 Tools*, and language interfaces (now for C and C++). O2 Tools is a front-end graphical programming environment, implemented as an O2 application. O2 Tools allows access to O2 utilities:

- The *O2 shell* provides facilities for dynamic creation and modification of a database schema as well as creation, compilation, and execution of CO2 programs. CO2 is an object-oriented extension of the C language for writing O2 application programs and method bodies. O2 Query is embedded in CO2, but may also be used independently as an interactive language.

- The *browser* for a schema and database.

- A *debugger* for the interactive maintenance of applications.

- The interactive mode of *O2 Query* for monitoring of the contents of the database.

- *O2 Look*, an interface generator tool. It is used to visualize, edit, and walk through the database schema and instances, including multimedia data.

O2 Look is the most powerful graphical tool to display complex objects we evaluated. It was implemented using the Motif toolkit. Any O2 object or value can be displayed on the screen with O2 using the *generic editor* , which also permits modification to displayed data. The generic editor associated with an object is a part of a *presentation* , which appears in a *virtual screen* (a resizeable window of the X-Window system). The presentation may be moved around on the screen, and has buttons to record changes, move the presentation to the foreground or background, or erase the display.

An editor associated with a complex object is composed of several subsidiary editors. The tuple editor, for example, which is the most commonly used, may contain any or all of the other editors. Each editor supports standard operations such as copying, cut-and-paste and erase. An editor associated with an O2 object contains a menu of standard entries of displaying and printing formats and public methods of the object. When users select attributes on the screen they automatically select an editor of the appropriate type. There are special editors for characters, booleans, tuples, list, sets, bitmaps, and text, each with unique additional functions and menus. In Look, the object structure is interpreted, so users can edit and rebuilt presentations without code modifications. O2 provides a set of primitives to manipulate object presentations. Primitives may be included in a CO2 application program to describe and manipulate the Look object presentation.

The C and C++ interfaces are implemented with an export/import of classes to/from O2. Applications written in C may use O2 to store their data as persistent. However, these interfaces are not as seamless for C++ applications as in ONTOS and ObjectStore. For example, in order to use C class Person into O2 a user must include *import from C* schema command:

```
import ''path/filename'' class Person from C;
```

A number of new C facilities would be available for the application from O2 as a result of the import implementation:

- *typedef* for O2_Person to declare C variables of the O2 class Person

- *Person_new*, *Person_read*, and *Person_write* functions to create O2 Person objects, read them into C structures and write C structures into them

Summarizing our presentation of the user interfaces and tools, we note that the ObjectStore user interface is oriented only to application designers, whereas ONTOS and O2 support interactive database processing as well. However, ObjectStore has special facilities to support low-cost migration of old programs.

4 Data models and persistence

The systems use the following terminology for a data description:

ONTOS	ObjectStore	O_2
type	class	class
property	data member	attribute
procedure	member function	method

ONTOS and ObjectStore use C++ Release 2.0 to describe program data, and extend it by adding facilities to provide object persistence.

In **ONTOS** all persistent objects are instances of some persistent classes. A separation of classes for persistent and transient is made at the level of a class description. All persistent classes must be derived from a superclass - the client library class Object. A database schema is defined by all persistent classes, each of them has a C++ class description. Classes in ONTOS often are named by types to provide a denotative class description. Types differ from classes in two aspects.

1. Types are denotable, i. e. it is possible to declare a variable as a pointer to a type. Denotable types, the representations of classes, enable an application to create new types at run-time.

2. ONTOS types may have *an extension* - the collection of all instances of the type, which is useful for writing queries.

Class descriptions are stored in header files and loaded into a database by the ONTOS *classify* utility. A persistent class description must satisfy the following criteria:

1. derivability from the class Object,

2. a special constructor member function (in addition to the usual C++ constructor) must be included. It is used to search for an object in the database and move it to an application program,

3. a *get Direct Type* member function must be included to return a pointer to the persistent object of the class in order to access it from a program,

4. if a class description has a destructor, then a *Destroy* function must be added to run when any exceptions are raised,

5. if a class description contains an operator *new*, then its signature must be identical to that which is used by ONTOS to allocate memory for a newly allocated object.

ONTOS supports object references of two types: direct references implemented as C++ pointers, and abstract references - instances of the ONTOS system class Reference. Abstract references allow a user to ignore whether an object pointed to by a reference is in memory or not. The method *binding* of the class Reference returns an in-memory pointer to the object, activating it (reading it from a database) if necessary. In general there are three main ways to activate objects in ONTOS:

1. using a system function to activate an object by name

```
Object* OC_lookup (char* objectName)
```

2. using a system function to activate an object by direct reference

```
Object* OC_direct Activate Object
                    (Entity** fieldAddress)
```

3. using an activation via an abstract reference

```
Entity* Reference::Binding
                      (Entity* context)
```

Abstract references also allow the use of different memory management techniques for different instances of a class. Each abstract reference is associated with a certain *context* - an instance of a special storage manager based class. When an abstract reference is created or reset to refer to a particular entity, its associated context stores a 32-bit value into the reference. This value denotes the entity. A binding method having a context as its argument allows the context to interpret that 32-bit value as a reference

to the original entity and return a pointer to this entry. Different contexts may implement different implementations of the same 32-bit reference value. We can see that abstract references take an extra indirection but improve flexibility.

Object names are organized in directories. Directories themselves are ONTOS objects. The ONTOS system *lookup* function provides searches of names in directories.

ONTOS supports a multiple inheritance as in C++ 2.0. There are some restrictions, however: 1) virtual persistent base classes are not allowed, 2) persistent base classes have to be public, 3) multiply inheritable classes can not be created dynamically.

Here is an example[1] of ONTOS class descriptions containing three classes:

- class Person with fields Name, Age, and Children,
- Person subclass Employee with field Employers,
- class Company with fields Name, Employee.

```
class Person: public Object{
 private:
  Char* name;
  Int age;
  Reference children;
 public:
  Person (Char* name);                    // 1
  Person (APL* theAPL);                   // 2
  ~Person ();                             // 3
  virtual Type* get Direct Type ();       // 4
  virtual void Destroy
         (Boolean aborted=FALSE);         // 5
  virtual void put Object
         (Boolean deallocate=FALSE);      // 6
  virtual void delete Object
         (Boolean deallocate=FALSE);      // 7
  Char* Name () { return name };          // 8
  Int Age () { return age };              // 9
  Set* Children () { return (Set*)
         children.Binding (this); };      // 10
  void Name (Char* new_name)
         { name=new_name };               // 11
  void Age (Int years) { age=years };     // 12
  void Children (Person* new_kid)
  { ((Set*) children.Binding
         (this))->Insert(new_kid);};      // 13
  };
```

[1] Examples in this paper are intended only to illustrate the general characteristics of syntax in the various system described, and have not actually been tested on the target systems. As a result, the syntax is not guaranteed to be totally accurate or complete.

```
class Employee: public Person{
 private:
  Set* inverse Company.employers employees;
 public:
  Employee (Set* Company);                // 1
  Employee (APL* theAPL);                 // 2
  ~Employee ();                           // 3
  virtual Type* get Direct Type ();       // 4
  virtual void Destroy
         (Boolean aborted=FALSE);         // 5
  };
class Company: public Object{
 private:
  Char* name;
  Set* inverse Employee.employees employers;
 public:
  Company (Char* name);                   // 1
  Company (APL* theAPL);                  // 2
  ~Company ();                            // 3
  virtual Type* get Direct Type ();       // 4
  virtual void Destroy
        (Boolean aborted=FALSE);          // 5
  virtual void put Object
        (Boolean deallocate=FALSE);       // 6
  virtual void delete Object
        (Boolean deallocate=FALSE);       // 7
  };
```

Apart from a normal constructor (1) an ONTOS class definition also takes a special constructor (2) used during activation. (3) is a destructor. (4) and (5) are obligatory special member functions. (6) and (7) are optional member functions to write/delete an object to/from a database. (8)-(10) are read accessors, and (11)-(13) are update accessors for particular properties. The children property in the class Person is described as an abstract reference, an instance of class Reference, that is why we cast it to Set* type in (13). Member functions (6) and (7) are inherited in the class Employee from the class Person.

ObjectStore has no special persistent class. Each object can be declared as persistent when it is declared. Classes created in an application are added to the schema during compilation. At run time, if an object is written to a database then its description is added to the schema. Schema information is stored in a separate ObjectStore database.

A search of persistent objects in the database is done via navigation from other persistent objects using data member pointers, or via queries performed over persistent collections. A retrieve of an initial persistent object, i. e. an entry point of the database, is accomplished through the use of either *database roots* or *persistent variables*. The Object-

Store class *database* has member functions: *create, destroy, open, close*. A database root can be created as a variable of the system class *database root* by using the database class member function *create root*. Another system function of this class, *find*, returns a database entry point. A persistent variable declaration with an initializer automatically creates a database root which is the variable name, and it has the value of the initializer as its initial value.

ObjectStore differs from ONTOS in that it doesn't require constructors in addition to those required by C++, nor does it require object access functions. ObjectStore also provides global overloaded persistent New and Delete functions.

Both ONTOS and ObjectStore support an inverse data member concept inherited from Vbase [Andrews et al. 87]. Data members can be declared as inverses of one another. Inverses model bidirectional links, and support referential integrity in an easy way.

Our example takes the following form in ObjectStore:

```
extern database *db1
class person
{
public:
 persistent <db1> set <person*> extent;
 char* name indexable;
 int age;
 set <person*> children;
 person (char* person_name, int years,
         set <person*> kids)
 {name=person.name; age=years;
         children=kids;
         extent.insert((person*)this);};
 ~person () {extent.remove (this)}
}
class employee
{
public:
 set <company*> employers
         inverse-member employees;
}
class company
{
public:
 char* name;
 set <employee*> employees
         inverse-member employers;
}
```

The Person name is indexable. This facility is provided explicitly in ObjectStore and in O2 System. ONTOS allows indexes only on aggregate classes. Member functions consist of a constructor and destructor. They use *insert ()* and *remove ()* functions to add/delete a new object to/from a repository, implemented in our example with the variable named *extent*. The extent of a class is a collection containing pointers to all its instances. It is described as a persistent data member. Persistent data members are a special kind of persistent storage. Class extents are used often as database entry points.

In the **O2 System** a database schema is described using O2 shell commands. A data model is based on three fundamental ideas - a *value, object,* and *name*. Values are entities of O2. Values can be atomic or have a complex structure and can be grouped into types. Complex values are represented by tuples, lists, sets, and their compositions. An object has a value, identity, and a set of methods. If a value acquires an identity and a set of methods, and is included into a class, then it is transformed into an object, i. e. the value is encapsulated by the class. An object itself is a pointer to access the encapsulated value. An object and a value can be made persistent by executing the *name* instruction. Since whole classes or separate objects can be declared persistent, the specification of persistence is as flexible as in ObjectStore. The separation of objects and values allows construction of data structures both inside and outside of the class system. Named values allow, in particular, creation of a set of values as a repository for all or some objects of a particular class.

O2 provides an interactive style of a schema creation. Users can interactively add applications, classes, methods, attributes, names (either for class objects or for instances of types), and programs. They can also bind arguments to methods, define bodies of a method with CO2 instructions, delete classes, methods, and instances, and make class attributes and methods private or public dynamically. They can run application programs and execute CO2 instructions. O2 programs are grouped by applications (named groups of programs) and are related to a particular database. A user can launch separate programs, or complete applications using the O2 shell.

A method body is implemented using the CO2 programming language. Since the actual choice of method (called binding) is done at run time, the schema of methods may be changed at any time without recompiling existing methods. Method versioning is allowed, and the *rename* command can choose a method version.

The O2 type system supports multiple inheritance. The common subclass can either define its own implementation of an ambiguous method (overwriting), or it can specify which superclass to inherit a method

from with the *from* qualifier .

Obligatory constructors and destructors, used in C++, are missing among O2 methods. Object initialization is the responsibility of the user describing the class. Adding and deleting objects to/from the database is done with the *add/delete name* commands. Late binding allows dynamic addition of new methods.

Our example in O2:

```
add class Person
 type tuple ( name: string,
              age: int,
              children: set (Person));
 method init (): person in class Person
                   is public;            //1
public read name, read age, read
                   children in class Person;
public write name, write age, write
                   children in class Person;

add class Employee
 inherits Person
 type tuple ( employer: set (Company));
public * in class Employee;

add class Company
 type tuple ( name: string,
                employers: set (Employee));
public * in class Company;

add name The_persons: set (Person);       //2
add name The_employees: set (Employee);   //3
add name The_parts: set (Company);        //4

execute CO2 { The_persons=set ();}$;
execute CO2 { The_employees=set ();}$;
execute CO2 { The_parts=set ();}$;

body init (): Person in class
                   Person CO2{            //5
 O2 Person new_person;                    //6
 The_persons += set (new_person);         //7
 return (new_person);                     //8
 }$;
```

(1) is an initialization method signature. Its body (5)-(8) is described using CO2 instructions. Each of the classes Employee and Company includes the *public** instruction, which makes its structure public. The attributes of the class Person are rendered public by the *public write* command.

In addition to the three classes, there are three *named values* (2)-(4), one corresponding to each class. The named values are used as repositories for the objects of the classes. Moreover, objects which are members of these named sets are persistent.

5 Aggregates

Each of the systems provide ample opportunities to develop data aggregates.

In **ONTOS** the aggregates are represented by the persistent class *aggregate* and its derived classes: *set, list*, and *association*. The last one itself has subclasses *array* and *dictionary*. The aggregate class defines some common properties for all subclasses: 1) *memberSpec*, which return a type of the aggregate, and 2) *cardinality*, returning the number of objects. Aggregates also specify a number of procedures: *isMember, isSubSet, checkMemberSpec, getIterator, getClusterSize, putCluster*. Each of the aggregates defines an *isSimilar* procedure, which checks if the members of two Aggregates are of the same class and are organized in the same way. The aggregate functions generally apply to a particular member or the entire aggregate. Functions for individual members include *Insert, setMember, Remove, the [ ] get member* operator , and *isMember*.

Each of the leaf aggregate classes - list, array, dictionary, set - has an associated nonpersistent auxiliary iterator class. Iterators query aggregates sequentially, usually over a user defined range. All aggregate iterators follow a similar protocol. Each of them defines a constructor, a *Reset* function, a *moreData* function, and a ( ) operator, returning the aggregate elements one by one. An iterator enables access to all or a specified subset of the aggregate elements.

The functions *copy* and *activation* handle entire aggregates. The copy constructor is defined for each aggregate class. Inactive elements are not copied by the copy constructor. That is why the copy constructor is more efficient for making copies than iterating over the aggregate and inserting the elements into a new aggregate one by one.

The *set class* is implemented with linear hashing to allow for growth in the number of set elements. There are insert and delete functions, the *SetIterator* class, a copy constructor, *isSimilar, isSubSet*, and deactivation functions *putObject, putCluster*. The set of functions is modest, but can be expanded very easy.

The *list class* has the *ListIterator*, insertion, updating, removal of elements and other function, specified in the class *aggregate*. Why was the Iterator is introduced? The efficiency of a *for* loop would have been far worse because it would have required starting from beginning of a list for each loop iteration. The reason is the C++ *for* loop does not know inter-

nals of the list object.

Associations - the *array* and *dictionary* classes - are classes whose every member is associated with a key or an index. Array indexes cover a continuous range of integers. Arrays can be resized by specifying new bounds. Dictionaries may be ordered or unordered and may or may not allow duplicates. *Ordered dictionaries* use B* tree access structures. *Unordered dictionaries* use a linear hashing algorithm.

ObjectStore provides three aggregate classes, called *collections: Set, Bag,* and *List*. The functions *insert* and *delete* are supported for Set and Bag classes. The ObjectStore *foreach* loop operator was added to C++ to search elements instead of the iterators in ONTOS. Elements of sets and bags are restricted to pointer types only for the current release.

The classes Set and Bag are parameterized with the types of their elements. The parameterized classes of ObjectStore have been approved as an ANSI draft standard version for a future C++ release.

The classes Set and Bag have four constructors:

- empty collection constructor
- copy constructor
- conversion constructor to transform a set into a bag and visa versa
- singleton constructor to create a collection with one specified value

The classes Set and Bag define a number of set-theoretical operations like a union, difference, intersection, and many others. Sets and bags can be mixed in these operations. Arguments may also be not only collections, but elements as well.

ObjectStore provides a variety of ways to control iteration order by describing a path expression on data members. Class instances are processed in order of the data member values. The data members mentioned in the path expression must be declared as *indexable*.

Updates performed within an iteration to the data member controlling the iteration order are dangerous. For example, if values of a data member, defining an access path of a *foreach* loop, are changed in the loop body, these values could be visited again. The same problem exists in ONTOS as well.

Adding or deleting *indexable* declaration forces a recompilation of the class declarations and a reorganization of all existing persistent objects too.

If the *foreach* loop doesn't provide sufficient control over the iteration process for some applications, ObjectStore supports access to elements of collections using a *cursor* facility. The cursor class includes member functions to create and move cursors, and retrieve an element by cursor. Cursors provide much more elaborate and flexible ways to retrieve elements of collections than does the foreach loop.

The **O2 System** supports three structured types: *sets, lists, and tuples*. The specification of a class contains three parts: the identifier of the class, the type specification, and a list of methods. A usage of the tuple type as the type specification makes the description of class attributes very convenient.

The most important set operations in O2 are:

- union, intersection, and difference
- addition, removal, and membership testing of elements
- filters - an extraction of a subset specified by a condition
- conversion of a list to a set
- iteration: *for i in X* {*instruction*}, where the variable *i* must be of the same type as the elements of X. A *when* modifier may optionally be added to restrict the iterations to a subset of X

Some interesting list operations are:

- concatenation
- append
- testing membership
- extracting and modifying elements
- extracting a sublist
- filters
- conversion of a set to a list

Lists, sets, and tuples can be integrated into more complex structures by arbitrary composition.

6 Query Facilities

An **ONTOS** user can write queries using C++ or Object SQL. The ONTOS C++ interface supports the *Instance Iterator* class which defines an iterator that yields all the instances of a given type. The usage of the instance iterator is possible only if the class is described as a type, that is it has an extension. The *Iterator* class is abstract (i. e. without instances) non-persistent ONTOS class, having a few derived classes - Array Iterator, Instance Iterator and others.

The ONTOS *lookup* function, allocated in the iterator body, activates a named object. Consecutive

executions of the iterator in the loop body return all instances of the given class. The search is performed by name on the hierarchical directory.

Object SQL is implemented with a single class called *Query Iterator*, which allows queries to be stored as objects in the database. Each instance of this class represents a particular query. The results of the query are obtained by calling the *yieldRow* member function. Each call returns the next row of results, like a FETCH statement of a relational SQL. A query has a usual "SELECT ... FROM ... WHERE" form. Query iterators support recursive and hierarchical queries. The FROM clause in Object SQL accepts any argument that evaluates to a collection of objects in addition to class names. The SELECT clause accepts property names as well as member function invocations and navigational-style property-chain expressions, like *person.children.children.age* for the grandchildren's age of our class Person.

An example of an Object SQL query using ONTOS C++ notation:

```
Entity *current_name
QueryIterator query=(
             "Select Children.name
              From Employee
              Where Employers="Ford");
while (Query.more Data ())
      { Query.yieldRow (current_name)
        <process data>
      }
```

Indexes in ONTOS are created with *Association Constructor* at compile time and can't be created or destroyed dynamically.

ObjectStore provides data navigation with standard C++ facilities, expanding them with an associative data processing facility - a DML query expression. The query expression has a form

expression 1 [:expression 2:]

where the expression 1 is a collection and the expression 2 bounds selected elements. Queries may be nested. Queries can process Bag and Set classes, but queries involving List class instances are not supported by the ObjectStore release 1.1.

A C++ loop could express the identical actions executed by the query expression, but the introduction of a special construction for queries improves optimization opportunities. A user can influence optimization with dynamic creation and deletion of indexes. The creation of an index on a complex path, i. e. on the path over a few layers of a data hierarchy, involves the creation and maintenance of indexes on each layer. The indexes are implemented with hash tables for unordered indexes, and with B-trees for range queries.

An example of a query in ObjectStore is

```
set <employee> selected_employee=employee
               [:employer="Ford":]
```

This query expression doesn't provide a selection of children's names. The user has to yield them by processing **selected_employee** in his C++ program, or can use a *foreach* iterator with a variable **a_path** as a parameter, where **a_path** defines a path of length 2

```
a_path = pathof (employee*, children->name);
```

The **O2 System** provides the CO2 language, an object-oriented extension of C, to describe both methods and queries. The O2 query language [O2 Technology 91c] may be used to write associative queries in CO2 programs as well. A query has an SQL-like form

select arg_1 *from* arg_2 *in*set_or_list_name
where condition

A query in CO2 always returns either a set value or a list value, and therefore it can be treated as a set or list value in the CO2 program or method. O2 queries may be integrated into CO2 or may be used autonomously as an interactive language. Only O2 supports queries against heterogeneous collections including lists and sets together.

In the O2 query language our example is:

```
select tuple ( e.children.name)
from e in Employee
where "Ford" in e.employers
```

7 Transaction models and concurrency control

O2 supports *conventional* transactions and performs concurrency control with a two-phase locking protocol on files and pages. Concurrent access to objects is handled by WiSS used as the low-level Object Manager layer [Velez et al. 89]. However, an application can not run more than one transaction at any particular time. The CO2 language includes only *commit* and *abort* commands for transactions. Restarts, checkpoints, and nested transactions are not provided.

Both ONTOS and ObjectStore provide a number of options to support a concurrent application execution. **ObjectStore** supports *conventional transactions* , including *nested transactions* , and *long transactions* .

Conventional transactions may be described in one of three ways:

- *transaction* statement - for applications using DML
- *cpp macros* - for applications using only the C++ library interface
- the above methods describe only static, lexical transactions. For dynamic transaction boundaries member functions *transaction::begin ()* and *transaction::commit ()* of the ObjectStore *transaction* class are used

The *transaction ::abort ()* function, executed within nested transactions, provides a way to abort the innermost transaction or the outermost one. However, no locks are released until the outermost transaction is terminated. Lexical transactions execute redo after system aborts, but dynamic transactions do not. Read-Only transactions may be declared to increase performance. Long transactions are used for no-conflict concurrency control to support the cooperative group model.

ONTOS implements both *conservative* (conventional) and *optimistic* concurrency control. With an optimistic policy readers and writers do not conflict, but it increases the risk of a preemptive transaction abort compare with a more conservative policy. The optimistic policy allows a read lock to be set on an object that already has a write lock if the conflicting transactions can be serialized (as if one transaction occurs entirely after another). The ONTOS *transactionCheckpoint* function can be used to provide additional points of time to serialize transactions. ONTOS supports two lock-resolution options: waiting until the lock is relinquished or conflict notification.

The ONTOS buffering policy allows:

- no buffering (immediate write)
- write is performed after about ten put operations (the default policy)
- write is performed either when the transaction is committed or the buffer is full

Cache cleanup functions may be executed after the commit of one transaction and before the start of another one. The options for a cache cleanup are:

- all objects are deallocated
- no cache cleanup
- objects are maintained in a form to be used by a following transaction
- selective cache refresh to reread only modified objects

ONTOS also has a nested transaction facility that is analogous to ObjectStore.

8 ObjectStore Cooperative Group Model

Today, with increasing application design complexity, it is important that application team members be able to create new objects or new object versions without overwriting someone else's work, as well as preventing new versions from use by others. In addition we would like to keep our temporary versions as private.

ObjectStore proposes very attractive facilities to provide *cooperative group work*. This feature is notably absent in the other systems. A user can check out groups of objects, make changes, and checks them back in to the main development branch. The changes may be visible to other team members. The cooperative group work supports *configurations* of objects that are treated as a single versioned unit, *workspaces* as environments for access and update of versioned object configurations, and *no-conflict concurrency control*. The same application can access both versioned and nonversioned instances of the same type. The virtual memory mapping architecture allows no penalty for access to non-versioned data.

The *new* operation has a *configuration* argument to allocate a new object into a configuration. Groups of objects can be placed into a single configuration and treated as a single unit for versioning. The configurations may be nested.

A user designates a workspace to control access to versioned data in configurations via an argument of *do_transaction*. Workspaces are created to perform a set of tasks. Multiple users can selectively control access to each other's work in progress through the use of nested workspaces. If a user has a workspace and configuration, he can *check out* the configuration into the workspace to work on it. It creates a new version, but the version is visible only from the user's private workspace. Transactions always operate in the context of the current workspace. When the work is done, the user *checks in* the configuration to make the new version visible in the parent workspace to other team members.

A versioned object can be changed only if its configuration is checked out. Checking out freezes the old version of every object in the configuration. But

Features	ObjectStore	ONTOS	O2
page server architecture	+	+	+
SQL-like interface	-	+	+
graphical schema designer	+	+	+
graphical browser	+	+	+
graphical data editor	-	+	+
debugger	+	-	+
C++ interface	+	+	+
easiness of existing C and C++ program migration	+	-	-
persistence at the level of objects rather then at the class level	+	-	+
metaclass support	-	+	-
indexing	+	+[2]	+
inverse data members	+	+	-
explicit object deletion instead of garbage collection	+	+	+
dynamic adding new classes	-	+	+
data aggregate support	+	+	+
query optimization (simplistic)	+	-	-
conventional transaction	+	+	+
nested transactions	+	+	-
long transactions	+	-	-
optimistic transactions	-	+	-
fault recovery	+	+	+
cooperative group model	+	-	-

Figure 1: Features of ObjectStore, ONTOS, and O_2

the workspace hierarchy is a dynamic structure, describing how users share the data of the configuration hierarchy.

No-conflict concurrency control is appropriate because users have private versions of configurations. Privately checked out versions are guaranteed conflict-free: they will never interfere.

9 Conclusion

Each of the systems discussed has a modern client/server architecture and offer a number of interfaces and tools for object-oriented application development.

We summarize particular system features in the Figure 1.

10 Acknowledgments

I am grateful to Alberto Mendelzon for starting me on the line of research presented here, to David DeWitt and Kurt Brown for their helpful comments on earlier drafts.

[2] ONTOS supports only static indexes for *associations*.

References

[Andrews et al. 87] T. Andrews, C. Harris. "Combining Language and Database Advances in an Object-Oriented Development Environment", *Proceedings of the 2nd OOPSLA* , 1987, 430-440.

[Atkinson et al. 89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Mayer, S. Zdonik. "The Object-Oriented Database System Manifesto", *Proceedings of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989, 40-57.

[Bancilhon et al. 90a] F. Bancilhon, P. Bridon, M. James. "The O2 Object-oriented DBMS", *Rapport Technique, Altair 58-90*, October 1990.

[Bancilhon et al. 90b] F. Bancilhon, W. Kim. "Object-Oriented Database Systems: In Transitions", *SIG-*

MOD RECORD , 19(4) December 1990, 49-53.

[Chou et al. 85] H.-T. Chou, D. DeWitt, R. Katz, A. Klug. "Design and Implementation of the Wisconsin Storage System", *Software - Practice and Experience*, 15(10), October 1985.

[Deux et al. 91] O. Deux et al. "The O2 System", *CACM*, 34(10) October 1991, 34-48.

[DeWitt et al. 90] D. DeWitt, P. Futtersack, D. Maier, F. Velez. "A Study of Three Alternative Workstation - Server Architectures for Object Oriented Database Systems", *Proceedings of the 16th VLDB Conf.*, 1990, 107-121.

[Kim et al. 89] W. Kim, F. Lochovsky (eds.). "Object-Oriented Concepts, Databases, and Applications", *Addison-Wesley (ACM Press)* , 1989.

[Lamb et al. 91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb. "The ObjectStore Database System", *CACM*, 34(10) October 1991, 50-63.

[Manola 89] F. Manola. "An Evaluation of Object-Oriented DBMS Developments", *GTE Laboratories, Technical Report TR-0066-10-89-165*, October 31, 1989.

[Object Design 90a] Object Design Inc. "ObjectStore User Guide", Release 1.0, October 1990.

[Object Design 90b] Object Design Inc. "ObjectStore Reference Manual", Release 1.0, October 1990.

[Ontologic 91] Ontologic Inc. ONTOS Developer's Guide", Version 2.0, February 1991.

[O2 Technology 91a] O2 Technology. "The O2 User's Guide", Version 2.2, May 1991.

[O2 Technology 91b] O2 Technology. "The O2 Application Designer's Manual", Version 2.2, May 1991.

[O2 Technology 91c] O2 Technology. "The O2 Programmer's Manual", Version 2.2, May 1991.

[Velez et al. 89] F. Velez, G. Bernard, V. Darnis. "The O2 Manager: an Overview", *Proceedings of 15th VLDB*, 1989, 357-366.

[Zdonik et al. 90] S.B. Zdonik, D. Maier (eds.). "Readings in Object-Oriented Database Systems", *Morgan Kaufmann Publishers*, 1990.