A Survey of Commercial Object-Oriented Database Management Systems

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by John Atkins of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, III of the Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of UHCL, RICIS, NASA or the United States Government.

SoRReL

West Virginia University

Software Reuse Repository Lab

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
(304) 293-3607 email: sorrel@cs.wvu.wvnet.edu

A Survey of Commercial Object-Oriented Database Management Systems

John Atkins

June 4, 1992

1 – Introduction

1.1 – The Relational Data Model

The object-oriented data model is the culmination of over thirty years of database research. Initially, database research focused on the need to provide information in a consistent and efficient manner to the business community. Early data models such as the hierarchical model and the network model met the goal of consistent and efficient access to data and were substantial improvements over simple file mechanisms for storing and accessing data. However, these models required highly skilled programmers to provide access to the data. Consequently, in the early 70's, E. F. Codd, an IBM research computer scientist, proposed a new data model based on the simple mathematical notion of the relation. This model is known as the Relational Model.

In the relational model, data is represented in flat tables (or relations) which have no physical or internal links between them. The simplicity of this model fostered the development of powerful but relatively simple query languages that now made data directly accessible to the general database user. Except for large, multi-user database systems, a database professional was in general no longer necessary. Database professionals found that traditional data in the form of character data, dates and numeric data were easily represented and managed via the relational model. Commercial relational database management systems proliferated and performance of relational databases improved dramatically.

However, there was a growing community of potential database users whose needs were not met by the relational model. These users needed to store data with data types not available in the relational model and who required a far richer modelling environment than that provided by the relational model. Indeed, the complexity of the objects to be represented in the model mandated a new approach to database technology. The Object-Oriented Model was the result.

2 – Overview of Object-Oriented Database Management Systems

Zdonik and Maier [Zd] trace the origins of object-oriented database management systems to the fields of programming languages, artificial intelligence and software engineering. The task of the database researcher was to "integrate the results [from these fields] to capture the essence of modern databases; to wit persistence, concurrency control, consistency and a query language." Applications of object-oriented databases seem to focus on data that is not easily represented by the relational model, e.g. graphics work stations, computer aided design tools, office automation systems, CASE tools, etc. At this point in the evolution of object-oriented database management systems, there does not seem to be a general consensus on the definition of what the "object-oriented data model" is but there does seem to be general agreement on the features expected of the model.

Central to the notion of object-oriented models is the concept of complexity. Zdonik and Maier [Zd] state that object-oriented database management systems "address the two sources of complexity: (a) facilities for data abstraction and inheritance and (b) features to model the properties, relationships and complex design of the data." Data abstraction is concerned with encapsulating the behavior of objects within the database. This is certainly a significant departure from the relational model where behavior associated with the data is embedded in application programs external to the database. Indeed, the ability to encapsulate behavior of the data within the database via a robust (computationally complete) database language compensates for one of the weaknesses of the relational model. In the relational model, the concept of "impedance mismatch" highlights the absence of compatibility of the query language of the database with the high level language in which database applications are written. As Zdonik and Maier [Zd] observe, "An object-oriented database management system extends the data manipulation language, DML, so that more of the application can be written in the DML."

Inheritance in an object-oriented database management system facilitates the modelling of the "IS_A relationship" in the database. It is not unusual to include a class of objects that is a "specialization" of another class of objects. We regard the first class of objects as a subclass of the latter class. It follows that the structure and behavior of objects in the parent class should apply to the objects in the subclass. Inheritance is the mechanism in the object-oriented model that supports sharing of structure and behavior. Moreover, inheritance facilitates code reuse because a procedure (method) in the parent class can apply to objects in a variety of subclasses. For example, the class Geometric_Objects might have a procedure Display which, without replication in subclasses of Geometric_Objects, applies to objects in the subclasses Circle and Rectangle as well as the subclass Square of the class Rectangle.

In the introduction to the Special Features section of [Zd], Zdonik and Maier note that "recursively defined objects occur in many application domains." The ability to represent complex objects in an object-oriented database provides the "features to model the properties, relationships and complex design of the data." This allows us to represent an entity as a single object in the object-oriented model, thus addressing limitations in the relational model. Zdonik and Maier [Zd] enumerate several advantages that accrue from the ability to represent data as a complex object; namely,

- arbitrary levels of structure
- manipulate the object as a single unit
- facilitate direct sharing of component subparts
- represent components of an object
- determine derived data from collections of the whole.

2.1 – Object-Oriented Terminology

The newness of the object-oriented paradigm has precluded a "universal vocabulary" for the subject but there does seem to be a growing consensus of agreement on some of the terminology. This agreement on terminology has been nurtured by a variety of papers, [At], [Ca] and [Zd], whose purpose was to provide a vocabulary foundation for the paradigm. We begin with a brief overview of object-oriented terminology with a discussion of each term.

Entities in the object-oriented model are represented as objects. Nelson, [Ne], defines an object "to be a self contained set of variables which can only be manipulated by a set of methods (procedures) defined exclusively for that purpose." Zdonik and Maier, [Zd], define an object as "an abstract machine that defines a protocol through which users of the object may interact." It is clear from these two descriptions that we cannot separate the notion of the object as data (its structure) and procedures or methods that apply to the object (encapsulation). Typically, objects have associated with them a collection of "instance variables" or attributes. These instance variables represent descriptive properties of the object. The relational model also supports attributes (or column names) as descriptive properties of the entities. The major difference between the relational model and the object-oriented model is that in the relational model, there is no facility for extending the collection of data types provided by the model. However, in the object-oriented model, the emphasis is on extensibility in the sense that new types can be generated recursively from atomic types. Objects have complex states that are "expressed by functions (messages or operations) that are defined for it. The values of these functions are other objects [Zd]." Thus, an object may reference other (complex) objects via functions defined specifically for that purpose. For example, a vehicle object may have an instance variable "engine" that references an engine object that has a very complex state itself that references other objects. In fact, Cattell [Ca] regards the ability of an object to have a complex state (which permits the entity to be represented as a unit) as a new feature of the next-generation of databases when he writes "Data that might span many tuples in a relational database can [now] be represented as one data object."

We have observed that we cannot separate the notion of an object from the procedures associated with it. These procedures are usually referred to as methods and are invoked as "messages" sent to an object which is then called a receiver. The methods define the behavior of the objects. This is another significant improvement over the relational model in that behavior, via methods, in the object-oriented model is "encapsulated" with the data directly rather than external to the data as in the relational model. In the strictest sense of encapsulation, access to the objects is only permitted via the interface provided by the methods defined for the objects. True encapsulation obviates the need to know the structure of the objects.

A class or type serves as a structural description of an object as well as a repository for both the methods and the instances of the class (the objects). Every object and method must belong to some class. The relationships between the classes defines the underlying structure of the database. If the class structure is restricted to a tree, we have single inheritance. If, on the other hand, the class structure can be an acyclic directed graph, then the system supports multiple inheritance. Systems which support multiple inheritance over single inheritance are much more flexible in terms of modeling the database environment. For example, Hull and King [HK] define a "World Travelers Database" in their paper on semantic data modeling which includes a class Person with two subclasses Tourist and BusinessTraveler. If a person is combining a business trip with a vacation, it would be rather difficult to represent this person in a system with single inheritance. In a system with multiple inheritance, we would simply define a subclass, WorkingVacation say, that was a subclass of both Tourist and BusinessTraveler. In the absence of conflicts, the objects in WorkingVacation would inherit both the structure and behavior of the two classes Tourist and BusinessTraveler. We shall address the potential for conflict in multiple inheritance systems in the discussion on inheritance.

Nelson [Ne] quotes Wegner as defining object-oriented as comprised of objects, classes and inheritance. Inheritance is the mechanism in which the structure and the behavior of a class is inherited by all subclasses. That is to say if class B is a subclass of class A and class A defines an instance variable X, then all objects in B automatically have an instance variable X without B explicitly defining X. Moreover, all the methods of A apply to objects in class B in the sense that an object from B can be a receiver of a message in A. In a single inheritance system, there can be no conflicts in terms of ambiguous names for both instance variables and methods. However, in a multiple inheritance system, there is significant potential for ambiguity. For example, suppose class C is a subclass of both classes A and B where neither A nor B are subclasses of each other.

Suppose further that both classes A and B have an instance variable X and both have a method P. The question then is how does C inherit the variable X and the method P? Snyder [Sn] suggests several resolutions to conflicts of this type. One solution is to redefine within the subclass the ambiguous instance variables and methods thereby completely avoiding the conflict. A second solution is to make the ambiguity an error unless it is inherited from a single class (perhaps via multiple paths). A third solution is to order the superclasses and select the ambiguous variable or method from the "first" class in which it appears. The simplest solution is that of the Postgres system which simply makes it an error to have an ambiguity and refuses to create the class for which the ambiguity would occur.

Part of the advantage that accrues from inheritance is the ability to leverage code written for a class. Code that may be used by objects of different classes is known as polymorphic code. One type of polymorphism is known as "Extension Polymorphism" [Zd] where code that applies to instance variables of objects in a class may also be applied to objects in any subclass by simply ignoring any extraneous instance variables defined by the subclass. This type of polymorphism is to be expected since (a) objects in a subclass inherit the structure defined in superclasses and (b) the class - subclass relationship represents an "IS_A" relationship which means that every object in a subclass is technically an object in the superclass and therefore the behavior for objects in the superclass is appropriate for the objects in the subclass. General polymorphism applies to code that is written once but applies to different abstract types. This code has the ability to determine the type of the receiver and react accordingly.

Snyder [Sn] observes that inheritance compromises the impact of encapsulation. The goal of encapsulation is to ensure that the structure of an object is completely hidden from external view and access to the object is entirely restricted to the methods defined for the class. The difficulty with this assumption is that with inheritance, the structure defined by the superclass is assumed by all its subclasses. This diminishes the ability of a designer to modify the structure of a class without impacting the subclasses.

Database design, whether in the context of the relational model or the object-oriented model, is an evolutionary and dynamic process. Commercial database management systems usually have excellent facilities for modifying the database schema without impacting existing application code. Thus, an object-oriented database management system must support schema evolution without significantly disrupting the current status. Ahmed, Wong, Sriram and Lagcher in [Ah] define four general classes of schema modification. These are:

- Changes to the content of a node: these changes include adding or dropping an instance or class variable or changing the name of an instance or class variable, changing the domain or the default value of a variable or changing the inheritance specification of an instance variable.

- Changes to a method: these changes include adding or dropping a method, changing the name of a method, modifying the code of a method or changing the inheritance of a method.
- Changes to an edge: these changes include making a class a superclass of another class, removing a superclass of a class or changing the specified order of the superclasses of a class.
- Changes to a class: these changes include adding a new class, dropping an existing class and changing the name of an existing class.

An issue not mentioned in [Ah] is the question of the status of existing objects in a class whose modification impacts the structure of an object. For example, if an instance variable is added to a class definition when that class has an extent, what is the role of that instance variable vis-a-vis objects currently in the class?

It is often the case that we need to temporarily return to the state of a database that it was in at a given point in time. Versioning in an object-oriented system supports this ability. Versions are a mechanism for "for recording the history of an object [Zd]." Typically, an instance x of a class T is actually a set of objects (called the version set) forming a partial order which represents the state of "the object" for a given interval of time. Reference to the version set returns a single object from the set. Configuration management is concerned with selecting a correct set of objects from the version sets that forms a consistent representation of the database.

The final term that is of interest is that of "object identity." Khoshafian and Copeland [KC] observe that object oriented programming grew out of the need to have the ability to "... distinguish one object from another regardless of the [object's] contents ..." This is the essence of object identity. An object is not identified by its set of attribute values as are the objects in the relational model. Rather, an object has associated with it an "object identifier" which is unique to the object and is assigned and maintained by the system. The ability to indirectly reference an object by its identifier rather than some collection of attribute values lies at the heart of complexity. For example, we might represent committees as objects in a class Committees. Two instance variables for these objects might be Chair and Membership. The value of the instance variable Chair is a reference (object identifier) to an object in the class Person. The value of the instance variable Membership may be a set object containing references (object identifiers) to objects from the class Person (the members of the committee).

3 – The Object-Oriented Data Model

Zdonik and Maier [Zd] identify specific features that are necessary for a system to be considered an object-oriented database management system. In general, an arbitrary database management system must:

- support a database structure that understands the underlying model. It must also provide a language for creating, updating, deleting and accessing the data
- support relationships among the objects. The language must be such that information represented via these relationships can be retrieved
- provide persistence for the data after the current session concludes and includes recovery of the data in the event of a system failure
- provide sharing of the data among concurrent users and provide a mechanism for ensuring that a transaction completes in its entirety or is aborted
- support databases whose size is not limited by main or virtual memory

An object-oriented database management system must include these five features as well as have the ability to:

- support object identity
- provide encapsulation
- support objects having a complex state

Although Zdonik and Maier do not include inheritance as a necessary feature of the object-oriented model, it is our opinion that it has become such a necessity that to omit it risks significant criticism from the database community. We therefore include inheritance as a necessary feature:

- support for both structural and behavioral inheritance

In addition to these features which we may regard as mandatory in an object-oriented database management system, there are several features that Zdonik and Maier [Zd] identify as frequently included with a system. These are the ability to:

- define integrity constraints at the time of data definition
- provide different security levels of access to the data
- query the database using a declarative data language
- maintain database descriptions within the database
- define views of the data

- provide a specific interface for the database administrator

We shall use both the necessary and desirable features of an object-oriented database management system as a framework for evaluating the systems under consideration.

4 – Focus of the Survey

The survey will address ten areas that serve to distinguish one system from another and should be of interest to those considering purchase of an object-oriented system. These areas are:

- Representational Ability
- The degree to which the system supports inheritance and data abstraction.
- Schema Evolution
- The ability of the system to facilitate and manage change in the underlying design of the database.
- Versioning
- The level of support provided by the system for maintaining the history of the objects.
- Query Facility

The query languages of object-oriented systems generally fall into two categories: (a) computationally complete languages that mimic object-oriented languages such as Smalltalk and (b) extensions to relational query languages such as SQL and Quel.

- Interfaces

The presence of high level language bindings for the system such as C++ or Smalltalk and the availability of tools for building applications.

- Performance Indicators

Performance has traditionally been the last hurdle in any emerging database technology. We shall report on benchmarks that are available for each system under consideration.

- Platforms

The hardware platforms on which the database systems can execute.

- Concurrency Issues

The support provided for concurrent users. We wish to determine the ability of the system to maintain data consistency during updates by multiple users.

- Distributivity

The level of support, if any, for distributed data.

- Price

The cost of the system by platform and by number of users as well as yearly maintenance cost.

4.1 – Gemstone

Gemstone is one of the earliest commercial object-oriented database management systems It was introduced in 1985 by Servio Corporation of Alameda, California. The current release is Release 2.5 with Release 3.0 soon to be available or perhaps already on the market as of this writing. Gemstone is an appropriate system to begin our survey as it is perhaps the simplest [Zd] of the object-oriented DBMS's.

We shall analyze and describe each system in our survey based on the ten areas described in the section on the focus of the survey.

Representational Ability

Gemstone's data manipulation language Opal is essentially Smalltalk-80 with extensions required to provide database functionality. Gemstone completely supports the class hierarchy with both structural and behavioral inheritance. Inheritance is single inheritance. Gemstone does not explicitly provide support for managing a class's extents. The developer must create objects in some subclass (Array, Set or Bag) of the class Collection to maintain explicit access to a class's objects. Otherwise it is necessary to create individual names or symbols for each object and store the symbol in a symbol dictionary. For example, suppose that we create a class Person. We must also create an object of the class Set, say SetofPersons, so that whenever we instantiate an object of class Person, we must immediately include that Person object in the object SetofPersons.

Methods are written in the programming language Opal and are encapsulated within the individual classes. Any object in that specific class or any subclass of that class may be the receiver of a message invoking that method. Access to and the ability to modify the state of an object must be via the methods defined for the class (or for a superclass).

The instance variables (both object and class) are essentially one of three types [Ei]. A named instance variable is a slot that may contain a value or a reference to another object, i.e. contain an object identifier. This corresponds to an attribute or column in the relation model. The named instance variable may or may not have a type associated with it. If it does, this provides a form of integrity constraint. The second type of instance variable is an indexed instance variable which corresponds to an array in a conventional programming language. Arrays in Opal may grow dynamically. The third instance variable is an anonymous instance variable which is restricted to objects of the class Collection in which "only membership is of interest [Ei]."

In general, we regard Gemstone's representational ability to be faithful to the object-oriented paradigm and hence quite good. The only criticism is that it does not support multiple inheritance whose absence can be somewhat stifling in terms of flexibility of design.

Schema Evolution

Gemstone fairs rather poorly in this category. It is possible to modify the schema of a Gemstone database but not without difficulty. The Product Overview [Ge] states that "... classes can be freely modified during the schema prototyping phase. Iterative modifications to the schema are permissible provided no instances have been created." In order to begin creating objects in a class, we must mark the class as invariant at which point it cannot be modified. Ahmed [Ah] observes that schema evolution in Gemstone is "... limited, [permitting only] ..definition of new classes at runtime, addition/deletion of class and instance variables, etc." Most changes require recompilation of the changes. Objects in "old classes" retain their structure and behavior and thus migration requires substantial developer intervention.

Versioning

Gemstone does not support versioning. In [Bu], the authors suggest that Gemstone does "support" versioning in Release 2.5. However, the authors seem to equivocate when they write that "... a mechanism, not a policy, is provided for versioning." This would suggest that the burden of providing a form of versioning rests with the developer.

Query Facility

As we have indicated, the query facility, Opal, is a database extension of Smalltalk-80. As an extension of Smalltalk-80, Opal is a computationally complete language which means that it is possible to encapsulate all of the behavioral aspects of the model within the database.

Gemstone provides a rich set of predefined classes and methods that can be used in ad hoc programs as well as in the methods. The central Opal construct is the block. A one-argument block is typically used as the argument in a message sent to a Collection object which then iterates the code within the block by assigning in turn an object from the collection to the single argument. In this way, objects can be individually referenced and tested from a class. For example, if we wished to access an object in the class Person with the name John Smith, we send the message Select to the SetofPersons object from the class Set as follows:

```
    SetofPersons Select: [ :n | n Name = 'John Smith'].
```

Note the use of the unary method Name defined for the class Person which returns the value of the instance variable Name for the object of Person referenced by the argument n.

Retrieval can be enhanced if we define indexes and use indexed associative access for our subclasses of Collection. The technique is to use paths rather than relying on sequences of messages. For example, suppose that we have a class Automobile with an instance variable Body which references an object in the class BodyManufacturer who has an instance variable Name. we can decide if a car object has a body by Fisher with

```
SetofAutomobiles Select: { :n | n.Body.Name = "Fisher"}.
```

Note the use of the { } pair rather than []. The pair { } indicates a selection block; its only difference with [] is increased performance. The alternative is the much slower option of sending a sequence of messages as

```
SetofAutomobiles Select: [ :n | n Body Name = "Fisher"].
```

Here, the message Body to the object argument n returns an object from the class BodyManufacturer to which we now send the message Name and compare the returned string with the literal "Fisher".

Gemstone supports two kinds of indexing for non-sequenceable collections (sets and bags). Identity indexing permits "testing of two objects without reference to the objects states [Ei]." In effect, this compares the object's identifiers. The second index option is equality indexing which is used to test the states of instance variable using the standard comparisons of =, <, <=, etc.

Interfaces

Gemstone has three high-level language interfaces; the Gemstone C Interface or GCI, the Gemstone C++ interface and Smalltalk-80 as well as Smalltalk/V 286. The GCI consists of a collection of C library routines. As reported in [Ge], "These functions facilitate the transfer of messages, objects and status between a [Gemstone session] and an application program." Thus, the C library provides access to the Gemstone database from any high-level language that supports C function calls.

The Gemstone C++ interface essentially provides database functionality for the C++ programming language. This is facilitated by a preprocessor provided by Servio.

The Smalltalk interface "consists of a set of class and method definitions that integrate the Smalltalk environment with the Gemstone environment [Ge]." Gemstone also provides a Smalltalk browser which provides an optional access to the database objects.

Gemstone provides facilities, called Gateways, that can reference data in other DBMS's. One collection of Gateways is the SQL Gateways. These SQL Gateways currently provide access to relational databases implemented in Sybase, Ingres, Oracle and Informix.

Servio also provides a suite of tools to facilitate typical DDL activities. The Gemstone Visual Schema Designer (GS Designer) "allows the user to create, modify and delete ... class definitions [via] ... a windowing environment [Bu]." A more sophisticated tool suite is available with Release 3.0. This application environment supports the development of forms that can interface directly to the database. The forms are actually stored with the database as part of the behavior of the database.

The usual interface to Gemstone is via Topaz which is a command line interface. It provides tools to manage the classes and methods. Topaz also provides a breakpoint debugger which permits inspection and modification of instance variables.

Performance Indicators

Gemstone reports that its database performance has improved substantially from Release 2.0 to Release 2.5. Benchmarks have been provided by Gemstone which provide transaction times for Gemstone tests but without comparisons to other systems. Performance for Release 2.0 was known to be quite poor. We report here the results of the benchmarks provided by Servio. The test was made on a database of approximately 20,000 objects which required 5 megs of memory. A random selection of 1000 objects required five seconds, a traversal by selecting one object followed by its connections to over 3000 other parts required five and a half seconds and an insertion of 100 objects with 300 additional connections for the new parts required three and a half seconds.

Platforms

Gemstone runs on all VAX platforms running VMS. UNIX platforms that are supported are the Sun SPARCstations, DECStations, IBM Risc 6000 and Sony NEWS. Both TCP/IP and DECNet protocols are supported.

Concurrency

Butterworth, [Bu], describes the concurrency control of Gemstone as " ... optimistic and implemented as shadowing. ... optimistic concurrency ... [means] that objects need not be locked." Optimistic concurrency control assumes that changes made by one user will not conflict with concurrent changes made by a second user. If no conflicts occur, the changes are committed to the database. However, if conflicts are detected, the entire transaction aborts and a new transaction is started. Needless to say, this is not a very realistic strategy for handling potential inconsistencies in the database. Release 2.0 introduced locking of objects undergoing modification which improved the ability of Gemstone to manage concurrent users.

Distributivity

Gemstone does not support a distributed environment. However, it does support a client/server architecture with applications running on a Macintosh II, 286 or 386 IBM compatibles or an IBM PS/2.

Price

The host plus a 4 user license is $20,000. Each additional user up to 10 is $2,000 per user, from 11 users to 20 users, the cost is $1,500 per user and from 21 users, the price is $1,000 per user. The annual maintenance cost is 15% of the current price.

4.2 – ORION / ITASCA

ORION is an object-oriented database management system whose development was initiated by the Advanced Computer Technology (ACT) Program at the Microelectronics and Computer Corporation (MCC). Originally, ORION was a research project designed to explore the "integration of object-oriented programming languages and high-end database applications such as AI, CAD/CAM and Office Automation Systems [Ah]." The first version or ORION was ORION_1 which was a single-user system was introduced in 1987. It was followed by ORION_1SX in 1988 which was developed for a client/server, local area network architecture. ORION_2, introduced in 1989, is a complete distributed system. However, ORION is only available to the corporate sponsors of MCC.

ORION is available to the general public under the name ITASCA. ITASCA is vended by Itasca Systems, Inc.

Representational Ability

ORION/ITASCA supports multiple inheritance with that has a single root, namely the class OBJECT. There are two distinct strategies for resolving name conflicts from multiple superclasses. The first strategy permits the user to linearly order the superclasses. The first superclass in which the conflicting name appears is the class from which it is inherited. This method is known as "Preferential Inheritance." By default, the order of the superclasses is the order in which superclasses are associated with a class. This ordering can be overridden by user action. In one case, the user can explicitly specify that an instance variable or method be inherited from a specific superclass whose choice might violate the inheritance dictated by the ordering.

A second strategy permits the user to specify that an instance variable or method be inherited form a specific superclass and then be renamed within the class. For example, a class might inherit an instance variable A from a superclass C which it renames A1 in the class. The class then could inherit an instance variable A from a second superclass D and retain the name A within the class.

In addition to supporting complex objects, ORION/ITASCA explicitly supports the "parts-of" relationship between objects. This is also known as composite objects. The support for composite objects is one of many strengths of the ORION/ITASCA system. The MIT Survey [Ah] reports that ORION contains "detailed semantics for defining composite behavior, schema, inheritance, locking, conflict resolution, object dependencies, etc." The MIT Survey lists some of the more prominent features supported by ORION and relating to composite objects. These are:

- Value Propagation: sharing of values of instance variables between composite objects.
- Clustering: composite objects are considered as a unit for clustering related objects on the disk

for faster access.

- Schema Redefinition: objects may be changed from a composite objects to non-composite objects and vice-versa.
- Locking: a set of 8 special locks are available for composite object locking.

ORION also provides support for shared-value instance variables or class variable (where all instances assume the same value) and for default-value instance variables (instance variables without an assigned value assume the default value).

Schema Evolution

One of the design goals of the ORION system was to facilitate schema evolution in an object-oriented database management system. ORION supports all of the types of schema evolution discussed in the section on schema evolution. In particular, ORION supports changes to a class or to a class's instance and class variables, changes to the (super and subclass) relationships between classes and changes to a node in the class lattice. Dynamic modifications to the schema can create ambiguities within in the database. To address potential ambiguities, ORION has a set of rules called "invariants of a class lattice" which are used to resolve ambiguities resulting from schema modification. Ahmed [Ah] lists some of these invariants as:

- Class Lattice Invariant: the class must remain a rooted, connected, directed, acyclic graph;
- Distinct Name Invariant: all instance variables and methods in a class must have distinct names;
- Domain Compatibility Invariant: the domain of an inherited variable in a class must be the same as that in the base class from which it was inherited.

Versioning

Another indication of the richness of the features available in ORION is in ORION's support for versioning. Hughes [Hu] provides an excellent description of ORION's versioning capability. ORION distinguishes between two types of versions: transient versions and working versions. Each type is characterized by the operations permitted on them.

A transient version can be modified by its creator, it can be derived from an existing transient version (at which point the original transient version becomes a working version) and it can be converted to a working version either explicitly by the user or implicitly by system action.

A working version cannot be update but may be deleted. Thus, it may be considered stable. A transient version can be derived from it if updating is necessary.

Versions are managed and manipulated in ORION by a collection of commands. These are:

- Create: creates an object which by its nature is transient;
- Derive: generates a transient version from an existing object; if the original object is transient, it becomes a working version;
- Replace: changes the contents of a transient version to that of a specified working version;
- Promote: changes a transient version to a working version;
- Delete: permits deletion of a specific version or the entire version hierarchy;
- Set_default: specifies the default version from the version hierarchy; the default may be a specific version number or the keyword "most_recent."

Query Facility

ORION's query language relies on the class/attribute hierarchy. The instance variables associated with a class have domains which may be other classes. In turn, these domain classes may have instances variables whose domains are classes, etc. As an example, consider a modification of a class/attribute hierarchy which appears in The ORION Papers [Or] given by:

```
Vehicle
  Owner ------------- Person
  Manufacturer        State ------- State
    |                                Name
    |
  Company
    Dealer ------------- Distributor
    Engine_Supplier       Sales_Volume
       |
       |
    Supplier
     Warehouse -------- Building
                          Size
```

The example has seven classes: Vehicle, Person, Company, Distributor, Supplier and Building. The domain of the instance variable (attribute) for the class Vehicle is the class Company, etc.

We now wish to obtain the Vehicle object where the owner is from "Texas" and the Manufacturer has a warehouse whose size value is over 10000 or the sales_volume of the dealer exceeds 800. The query is:

```
Select Vehicle ( Path Owner State Name = "Texas") and
((Path Manufacturer Dealer Sales_Volume > 800) or
        (Path Manufacturer Engine_Supplier Warehouse Size > 10000))
```

Note the extensive use of the Path construct which essentially parallels the class/attribute hierarchy.

Interfaces

ORION is written in Common LISP. Originally, ORION suffered from performance problems since it required LISP as an underlying system. However, ORION is now compiled which has substantially improved performance. ORION interfaces to LISP. ITASCA interfaces to LISP and C. Ahmed [Ah] reports that "ITASCA also supports dynamic loading of FORTRAN code into its methods."

Performance Indicators

Explicit benchmarking was not available for ITASCA; however, ITASCA at one time needed interpreted LISP to execute which substantially degraded performance. The most recent version of ITASCA now has a compiled version which negates the need for an interpreted LISP and consequently provides significant improvement in its performance.

Platforms

ORION runs on Symbolics 3600 LISP machines and on SUN3 workstations. The MIT Survey [Ah] reports that ITASCA runs on "most UNIX systems including Sun, Apollo, Hewlett-Packard, Silicon Graphics, DECStation ..." and Data General"

Concurrency

ORION supports concurrency by " ... attempting to serialize transactions, i.e. it completely isolates a transaction from the effects of other concurrent transactions [Ah]." This simplistic approach has the effect of isolating transactions which inhibits data sharing and cooperation between transactions.

ITASCA [It] also implements serialized transactions but permits "simultaneous, independent transactions to execute in parallel. This allows transactions to request a lock on concurrent updates on the same object." Deadlock detection provides for notification to the user indicating a deadlock condition has occurred. The user may abort or resubmit the transaction.

ITASCA supports two types of transactions: short and long transactions. Short transactions are the classical transactions that must be completed as a unit or are completed aborted. Long transactions apply to private databases where an object is "checked out" to the private database and may be retained by the private database for long periods. When a checked out object is returned to the shared database, it creates a new version of the object and the long transaction terminates.

ITASCA also supports the notion of a shared transaction where "multiple clients may participate in the same transaction for cooperative work [Ah]."

Distributivity

ITASCA vends a distributed environment for UNIX workstations. Data integrity is ensured since ITASCA incorporates the two-phased commit for distributed updates.

Price

ITASCA Systems, Inc. reports a price of $3,995 for the first client/server license for most popular UNIX workstations. The supported workstations for the client/server environment are those listed under the section on platforms.

4.3 – G-BASE/GTX

G-BASE/GTX is a product vended by Object Databases located in Cambridge, Massachusetts. Object Databases is a subsidiary of Graphael of France. The documentation provided by the company was promotional in nature and lacked technical substance. However, we shall summarize some of the features of the product pair that were contained in the vendor supplied literature. We should note that this product was not mentioned in any of the journal articles obtained for this survey.

Representational Ability

GTX is a high performance data storage manager which is capable of supporting up to 16 gigabytes of data. The system is intended to facilitate storage and access of multimedia data such as voice, graphical images and video objects.

The vendor indicates that G-Base "supports an object-oriented extension [Ob1]." The promotional literature states that "G-Base offers the advantages of encapsulation through the use of methods, through the use of ... triggers and validation functions. Methods are defined for classes and can be inherited [Ob1]."

In the absence of detailed documentation on the G-BASE product, it appears that the product is really an interface between GTX and a variety of systems, both those that support the object-oriented paradigm as well as the traditional relational model.

Schema Evolution

The product announcement contains a single sentence relative to schema evolution. "The schema is dynamically modifiable by applications without recompilation, allowing schema evolution in a real time environment [Ob1]."

Versioning

GTX (and not G-BASE) contains an "intrinsic versioning" mechanism that automatically creates new versions of an object. When an object is updated, a new copy of the object is created (the updated object) while the system maintains the original object in its former state. The user can execute queries "as-of current or historical time [Ob1]."

Query Facility

No documentation available or provided.

Interfaces

The vendor supplied information seems to indicate that G-BASE interfaces easily to a wide variety of languages and databases; however, specifics were notably lacking.

Performance Indicators

Bench marks were not available; however, the product descriptions indicated that GTX contains appropriate software to monitor performance such as transaction timing and disk utilization.

Platforms

GTX runs on a VAX/VMS system while G-BASE is designed to run on the client platform. We assume that clients may be a variety of PC's although specifics were again lacking.

Concurrency

The GTX transaction model supports "two-phased locking [Ob1]" which we interpret to mean the standard two-phase commit protocol. GTX also supports disk replication in the sense that data is replicated on multiple disks to ensure that a single disk failure does not interrupt database processing.

Distributivity

The database environment of the G-BASE/GTX system is designed to have the GTX component run on the VAX mainframe while G-BASE resides on a client machine.

Price

The price is $100,000.

4.4 – Versant

Versant is vended by Versant Object Technology Corporation of Menlo Park, California. It is a database management system that provides database capabilities such as persistence and transaction management to C++ classes. The MIT Survey [Ah] rates this system as particularly well suited to engineering applications. The Concurrent Engineering Research Center of West Virginia University recently selected Versant as their object-oriented DBMS because of its strong interface to C++.

Representational Ability

Versant supports the multiple inheritance as implemented in C++2.0. Name resolution conflict resulting from multiple inheritance is resolved within C++. The Versant documentation does not specifically address the resolution of this problem. Indeed, there does not seem to be a standard within the C++ community at this time addressing the problem of name conflict resolution. It seems that C++ will return an error for class definitions where name conflict occurs. However, a class definition may resolve a potential naming conflict by qualifying the offending instance variable within the class by the ancestor class from which the instance variable is to be inherited. Some C++ compilers provide more complex rules for disambiguating functions that are to be inherited and whose names appear in multiple ancestor classes.

Schema Evolution

Versant provides a very limited schema evolution capability. The Versant System documentation [Ve2] states that, "Versant allows [the user] to create, rename or drop only leaf classes. You can also create, drop, rename or change individual attributes and methods in leaf classes." The MIT Survey [Ah] further reports that "... a change to classes other than the leaf classes will cause the class and all its subclasses and instances to be dropped." It also observes that "A deletion deletes all subclasses and instances."

Versioning

Versant supports versioning of objects. The user determines which objects are to be versioned and then uses a specific C++ interface to "prepare" an object for versioning and then to "create" a version of the object. One application of versioning is to permit a user to "checkout" an object to a personal database irrespective of locks that might apply to the object. When the object is "checked into" the group database, it automatically creates a new version. Object versions form a lattice structure in the sense that two or more parent versions can be merged to form a new version.

Versant maintains a default object version which is that version with the most recent version.

However, an application program can statically bind to a specific version since each version has its own object identifier.

Query Facility

The query facility for Versant is the C++ interface.

Interfaces

Versant provides direct support for C++, C, Smalltalk and Object SQL. With the C interface, languages, such as Fortran, COBOL or Ada, that can link to C via function calls can access Versant databases. Versant also supports a browser similar to that provided by Smalltalk.

Object SQL is an extension to the relational query language SQL that permits access to object-oriented databases. The syntax is the standard SELECT ... FROM ... WHERE. Hughes [Hu] describes the use of Object SQL (OSQL) where he writes that OSQL "uses object names in the FROM clause and properties in the SELECT clause. Using the familiar 'dot' notation, it is possible to retrieve information from related objects."

As an example of the use of OSQL, suppose we have a class EMPLOYEE with instance variables Name (character), Department (class Department) and Job (class Job). If we want the Department name of all engineers, we might pose the following query:

```
SELECT E.NAME, E.DEPARTMENT.NAME
FROM E IN EMPLOYEE
WHERE E.JOB.TITLE = 'ENGINEER'
```

Note the use of the 'dot' notation where we use the instance variable Name in the class Department and the instance variable Title in the class Job. An essential difference between the classical definition of SQL and OSQL is the use of the object itself in the query rather than key values.

Performance Indicators

The Versant product brief [Ve1] reports that Sun Microsystems conducted benchmarks on Versant with three unnamed object-oriented DBMS's, a relational DBMS and Sun's indexed file system. The result, as reported by Versant, is that Versant out performed all of the tested systems.

Platforms

Versant was originally implemented on the Sun3 and has been ported to the Sun4, the IBM RISC System/6000, the DECStation, HP 9000/400, Silicon Graphics IRIS Series and the Integraph 6000 Series.

Concurrency

The MIT Survey [Ah] reports that Versant has a "reasonably powerful transaction management" capability. The Versant Systems Manual [Ve2] provides an overview of their transaction management capability. Versant implements two-phased commits. In essence, Versant supports the notion of a short transaction which is a "set of related database actions that [are to be] saved or abandoned as a group." Long transactions "are only relevant if [the user] chooses a personal database for [the] session workspace."

The Versant Systems Manual reports describes four lock modes. These are:

- Null lock: permits viewing of objects regardless of current locks.
- Read lock: permits viewing an object without risk of a current modification of the object.
- Update lock: guarantees the next write lock on an object.
- Write lock: guarantees the right to read and change an object.

Distributivity

The Versant Brief [Ve1] reports that Versant "was designed from the start to operate in a fully distributed manner." Furthermore, "Versant's distribution is independent of hardware and software platforms - Versant automatically compensates for any differences among these platforms."

Price

$7,200 for the Database browser, $1,200 for the UI kit, $800 for Object SQL and $2,400 for the Object Modeler.

4.5 – ONTOS

ONTOS was developed by Ontologic, Inc. of Billerica, Massachusetts. It was initially known as Vbase. Its original intent was to "provide a complete development system for practical production applications based on object-oriented technology [An]." One of its goals was to integrate an object-oriented language with database functionality. Although the original object-oriented language was not C++, the current version of ONTOS uses C++ version 2.0 as the query facility for the system. In essence, just as with Versant, ONTOS is C++ version 2.0 with database capability. The MIT Survey [Ah] regards ONTOS as a rather poor product when they write that "ONTOS is probably the least suitable for collaborative application development, and offers almost none of the advanced features discussed [in the survey]." The survey also observes that ONTOS provides no security for the system.

Representational Ability

ONTOS supports multiple inheritance through C++. The resolution of name conflicts is handled within C++. Refer to the discussion of name resolution conflict in the section on Versant.

Schema Evolution

Schema evolution in ONTOS is very limited. Class variables or methods may be added or dropped only if the class has no instances. Changes to class variable will be propagated but class methods may not be changed at run-time. Inheritance may not be modified at run-time. Leaf classes are the only classes that may be added.

Versioning

ONTOS does not support versioning.

Query Facility

The query facility for ONTOS is C++ version 2.0. ONTOS provides a library of C++ utility classes for application development [Ah]. ONTOS supports an explicit exception handler from which the developer may specify actions to take in the event of an exception. ONTOS also provides a trigger mechanism which executes when a specified condition is met.

ONTOS also supports an OSQL interface which permits SQL queries to the database as described in Hughes [Hu] and in the discussion of the Versant OODBMS. Queries may also be formulated "in C++ using low level access and iterator functions provided by ONTOS [Ah]."

Interfaces

The MIT Survey reports that ONTOS provides "a graphical schema designer (called DBDesigner) developed using the X-Motif toolkit."

Performance Indicators

No information available.

Platforms

ONTOS currently runs on the UNIX environments for Sun3 and Sun4 and Apollo. It is being imported to VAX/VMS and OS/2 platforms.

Concurrency

ONTOS supports four lock mechanisms. The writeintent lock is used for objects that may be modified. The nolock permits reading of objects without locking the objects. The write lock is only used internally by the server. Read locks permits read access only.

Distributivity

Not supported.

Price

$3500 to $4000.

4.6 – ObjectStore

ObjectStore is vended by Object Design, Inc. of Burlington, Massachusetts. The design goal of this system was to "provide a tightly integrated language interface to the traditional database features of persistent storage, transaction management, distributed data access and associative queries [La]." The database is usually accessed via C++ but Object Design provides a C interface. The MIT Survey regards ObjectStore as a better system than ONTOS or Gemstone (but seems to prefer ITASCA overall). Our view is that this product seems to be one of the better commercial systems, second only to ITASCA but on a par with Versant.

Representational Ability

ObjectStore supports multiple inheritance via its C++ (version 2.1) interface. It also supports the notion of composite object by providing a specific class Parts to implement the Part_Of relationship by storing object identifiers of the parent part and the child parts. ObjectStore additionally supports the concept of an inverse relationship. That is, we can define a relationship between the class Body (of an automobile) and the class Door where objects in the class Door are 'Part_Of' the objects in the class Body. Similarly, ObjectStore provides facilities for establishing referential integrity constraints between classes in the sense that every Door object must be associated with a Body object (and conversely). ObjectStore will also permit the definition of inverse relationships between objects. As above, every Door object is Part_of a Body object and the inverse relationship that we might define is that every Body object Contains_as_a_part a Door object. With relationships such as these, whenever one object is deleted, the system removes the object identifiers involved in the inverse relationship so that dangling pointers are avoided.

Schema Evolution

ObjectStore provides a rudimentary schema evolution facility via its graphical interface. Instance variables and methods may be added, dropped or modified if the class has no instances. Leaf classes may only be added. No other changes are possible.

Versioning

ObjectStore's versioning mechanism is very similar to that provided by ITASCA (and hence is very powerful). ObjectStore supports the ability to check an object out of a shared database to a private database (to begin a long transaction). When the object is checked back into the shared database, it is done so as a new version (of the object).

ObjectStore also supports configuration management. Ahmed [Ah] describes ObjectStore's ability to manage configurations as "instances of ObjectStore's kernel class Configuration. [Configurations] typically consist of component objects that make up a composite design object or objects that evolved together. Configurations can be nested [within parent configurations]."

Query Facility

ObjectStore was implemented with a mix of C and C++. It supports AT&T's C++ version 2.1. ObjectStore also extends C++ by providing a persistent version of the C++ New message. It also provides several new C++ operators not originally available in C++ that enhance C++'s database capabilities.

In addition to supporting access to the database via C and C++, ObjectStore provides its own query facility via associative queries. These associative queries are used within the C++ interface and take advantage of ObjectStore's query optimization facilities. Typical applications of associative queries are provided in [Ob2]. For example, to create a collection Teenagers from the collection People, we might pose the following query:

```
set<Person*>Teenagers = People[: age>=13 && age <= 199 :];
```

We can also follow, in one sense, the 'dot' notation in these associative queries. Suppose that the class People has an instance variable Children which contains People objects. To find all children at home, we pose the following query:

```
set<Person*>Kids_at_home = people[: children[: age < 5 :] :];
```

Interfaces

In addition to the C++ and C interfaces, ObjectStore also provides "The SchemaDesigner" which is "an interactive schema design tool for graphically developing, viewing and evolving [the schema] [Ob2]." ObjectStore provides a "Browser" which is a graphical mechanism for inspecting the contents of an ObjectStore database. There is also a debugger available.

Performance Indicators

Lamp [La] reports on an unpublished benchmark provided by Cattell. The results of one test were provided in the document. The result was that ObjectStore had the lowest traversal time among four unnamed object-oriented systems, a relational database system and an indexed file system.

Platforms

ObjectStore runs on Sun3 and Sun4 running OS, a Solbourne OS/SMD, the IBM RISC System/6000 running AIX and a DECStation running ULTRIX. Object Design reports that ObjectStore is being ported to UNIX System V.4, Hewlett Packard HP/UX, Microsoft Windows 3.0 and OS/2 2.0.

Concurrency

ObjectStore supports two locking mechanisms: read-only and update locks. The read-only locks obtain a snapshot of the database but don't permit updates. Update locks place write locks on the specified data.

As in ONTOS, ObjectStore provides an exception handler to manage transaction conflicts or an abort.

The two-phased commit is enforced for distributed databases.

Distributivity

ObjectStore supports a distributed system. In this system, multiple ObjectStore databases may reside on diverse platforms as well as on the client workstation. The literature provided by Object Design reports that "ObjectStore's distributed architecture is based on TCP/IP networks for interoperability among popular UNIX-based systems."

Price

$7000 for a 2 user license and $4200 for the DML

4.7 – Objectivity

Objectivity is an object-oriented database system that is another database extension of the C++ programming language. The product is vended by Objectivity, Inc. of Menlo Park, california. The only documentation obtained for this study was provided by the vendor. It consists of several glossies and a System Overview manual [Ob3].

Representational Ability

Since Objectivity is an extension of the C++ programming language, it provides for multiple inheritance. The question of name conflict resolution again depends on C++ and is not a specific database issue. Objectivity, in the System Overview, emphasizes the "association" feature of the system. This feature provides an opportunity to define relationships among the objects. The relationships may be unidirectional or bidirectional and may be many-to-many (as well as many-to-one). It appears that this is just the ability to support complex objects via defined relationships. The System Overview notes that bidirectional associations provide for referential integrity.

Schema Evolution

No information was provided in the documentation; it is assumed that schema modification defaults to that available in C++.

Versioning

Objectivity supports versioning. It provides two options for versioning; linear versioning requires that a new version be created from the most recent version of the object. Branching version supports the creation of a new version from an arbitrary version of an object. We can merge versions thereby creating a lattice of versions. When a new version is created, there are three options that are possible relative to the objects associations. The association may be dropped for the new version in which case it remains with the old version. The user may copy the association so that both the new and old versions are associated with the same object(s). The user may move the association so that the association is now with the new version and the old version has a null association.

Query Facility

The query facility is that of C++. The C++ language features are enhanced by Objectivity's inclusion of dynamic arrays and associations.

Interfaces

The language interfaces are C, ANSI C, C++ versions 2.0 and 2.1. Objectivity provides a rich collection of graphical interfaces to the database; these are:

- Graphical Tool Manager: used to manage database tools;
- Graphical Data Browser: used to select and view database object;
- Graphical Type Browser: used to view database schema;
- Debugging Utility: used to dynamically inspect database objects;
- Concurrency Monitor: used to monitor database access across the network; may be used to clear or set locks;
- DBA Utilities: used to administer the components of the database.

Objectivity also supports X-windows as well as an ICCCM compliant window manager. The schema is defined via Objectivity's own Data Definition language which is essentially that of C++.

Performance Indicators

No information provided in the vendor supplied documentation.

Platforms

Objectivity runs on the following platforms; DECStation: Ultrix 4.1, VAX running both Ultrix 4.1 and VMS 5.4, HP 9000 series 300,400,700 and 800, Hewlett Packard running UX 7.0, an IBM PC (runtime version only) under DOS 3.3 and Windows 3.0, an IBM RISC 6000 running AIX 3.1, SONY NEWS running NEWS-OS 3.3 and the Sun3 SPARCstation running SunOS 4.1. Objectivity has recently included Silicon Graphics Workstation running Iris and NCR running UNIX.

Concurrency

Objectivity supports two locking mechanisms: a read lock and an update lock. The read lock permits other processes to read the data but not update it and an update lock prevents all other processes from reading or updating the data.

Objectivity also supports long and short transactions. The long transactions correspond to the ability to "checkout" an object. Usually, an object is checked out to a private database. In Objectivity's case, checking an object out corresponds to placing a lock on the object that endures past the current session. A "checkin" removes the lock.

Distributivity

Objectivity supports "peer-to-peer [Ob3]" architecture. It takes advantage of workstation computing power in that it supports a client/server architecture.

Price

Objectivity has three pricing options:

- Database Development System (DDS) at $18,000 with maintenance at $3,600 per year.
- Application Development System (ADS) at $9,000 with maintenance at $1,800 per year.
- Runtime System at $400 per unit with maintenance at $25 per unit per year.

4.8 – The O_2 System

The O_2 database management system is vended by O_2 Technology, located in Versailles, France. O_2 Technology has an American office in cambridge, Massachusetts. The vendor's literature [O2] describes O_2 as "an open object-oriented database management system integrated with a complete application development environment and a set of advanced graphic user interface development tools." O_2 was designed as a true object-oriented database management system rather than providing database functionality to an existing object-oriented programming language such as C++. Indeed, O_2 states that its design addresses three essential ideas:

- The application of object-oriented principles to the database environment;
- The fusion of a graphical user interface (GUI) programming language and database technologies and
- Conformance to standards.

O_2 has thoroughly integrated the entire database environment; it has avoided the impedance mismatch which often plagues database applications by representing objects "in an identical fashion on the screen, in memory and on the disk [O2]." The result is a simpler development environment and reduced maintenance time. O_2's development environment is replete with several products that enhance application development, provide easy ad hoc access to the database and support integration with existing C and C++ applications.

Representational Ability

The O_2 data model provides the ability to define complex objects using primitive types. The primitive types are boolean, character, integer, real, string and bits. Complex types can then be developed using the tuple, list and (unique) set type constructors. A list is analogous to an array with indexable entries. A set behaves as a bag in other systems; with the unique qualifier, it becomes a set in the traditional sense. The tuple constructor is much like a record structure in conventional languages. We illustrate the tuple and other features of O_2 by examples which appear in [O2].

```
type monument:
tuple(name : string,
address: tuple(city: City,
street: string,
number: integer),
  admission_fee: real,
  statistics: list(tuple(date: Date,
  number_visitors: integer));
```

A value for a variable of type monument has six components: a name of primitive type string, an address whose value has three components, an admission_fee of primitive type real and statistics whose value is a list of values each of whose values is a pair consisting of a date and a count of visitors. Note that the type of the instance variable city is the class City. We can now create an object of type monument and assign a value as follows:

```
run body{
o2 monument tower; /*Create an object of type monument*/
/*Now assign values to the components of object tower*/
tower.name = "Eiffel Tower";
tower.address = tuple(city = Paris,
               street = "Champs de Mars",
               number = 1);
tower.admission_fee = 42.50;
tower.statistics += list(tuple(date = today(),
                                        number_visitors = 9710));
```

The value for the instance variable city is an object of class City, namely Paris. The name Paris does not represent a string but rather an object that is referenced by the name Paris. The notation += for the assignment for statistics is a dynamic allocation of the next position in the list for statistics.

In fact, the class City can be defined by:

```
class City
type tuple(name: string,
      map: Bitmap,
      hotels: set(Hotel))

method how_many_vacancies(star: integer): integer,
   build_new_hotel(h: Hotel)
end;
```

Object of class City have three components: a name, a map of the city and a set of hotel objects (from the class Hotel). Two methods are defined for the class. The method how_many_vacancies has one parameter which reflects the request to determine the number of vacancies in a 5 star, say, hotel. The method returns an integer which represents the number of vacancies in hotels whose rating is given by the parameter value.

Name conflict resolution is resolved by explicit renaming within the class definition. For example, suppose that we have a class Hotel with instance variables Name, Stars and Free_rooms and Restaurant with instance variables Sign, Stars and Menus. We define a class Hotel_Restaurant that inherits from both Hotel and Restaurant as follows:

```
class Hotel_Restaurant inherit Hotel, Restaurant
```

```
rename starts_Restaurants as forks,
   starts_Hotel as stars
end;
```

Note that we inherit both instance variables stars from Hotel and Restaurant and simply rename them within the class Hotel_Restaurant.

Objects are made persistent by providing the object with a name. For example, the object whose name is Paris is made persistent with the command:

```
name Paris: Tourist_City;
```

Any object attached directly or recursively to the object paris also persists.

Schema Evolution

The documentation [O2] briefly discusses schema evolution as part of the schema manager. Essentially, classes may be incompletely specified. A class can be defined with an instance variable whose type is a class not yet specified. However, the body of a method may not refer to incompletely specified classes or methods. On the other hand, deletion is more well defined. A class may not be deleted unless it is a leaf class. If the deleted class appears in the signature or body of a method, that method's body is invalidated. More specific information concerning schema evolution was requested from the vendor. Their response was "What we have now is the capability to change class and methods definition and implementation. However, we do not guarantee the consistency of the class instances. We are currently working on this consistency problem and also on the integration between schema evolution, database reorganization and versioning (so that one can version together the old schema with the old objects)." In terms of the discussion of the goals of schema evolution as defined in the MIT survey and the capability provided by ITASCA, O_2's capability is limited.

Versioning

Versioning seems to be rather limited as well in O_2. It is mentioned in [De] with " O_2Tools [the O_2 tool set] manages versions of bodies of methods and programs." No mention is made of versioning of objects or configuration management. Versioning is not discussed in the technical documentation. However, we requested additional information on versioning from the vendor. Their response was "We provide a versioning mechanism for objects based on the inheritance from a system defined class. This class provides methods to create versions and to retrieve them. This class can of course be specialized to adapt specific environments."

Query Facility

O_2's query facility, O_2Query, is very much like object-SQL. It is a subset of one of O_2's tools,

O_2C, but may be used independently for ad hoc, interactive access to the database. It may also be used as a function call from C or C++. All O_2 data types, operators and methods are permitted inside a query. As an example, we wish to find all restaurants in Paris where we may eat for less than 100 Francs. The query is:

```
select tuple(restaurant: r.name,
choices:  select tuple(price:menu.rate, food: menu.contents)
 from menu in m)
from r in restaurant, m in r.menus
where r.address.city.name = "Paris"
and (exists menu in m: menu.rate < 100)
```

Interfaces

O_2 has an excellent collection of interfaces and this perhaps distinguishes it from its competitors. First, O_2 provides both a C and C++ interface. O_2 has an export capability which supports the generation of C++ classes from O_2 from which the O_2 data can be manipulated via a C++ application. Conversely, C++ classes can be made persistent with the import facility. Any feature of O_2 is accessible directly from C++ or C.

In addition to the C and C++ interface, O_2 provides three tools within their O_2Tools collection; namely O_2C, O_2Query and O_2Look. O_2Query was discussed in the query language section.

O_2C is the language in which the bodies of the methods are written. Indeed, O_2C is a computationally complete programming language. As an example of an O_2C program, we include the body of the City method how_many_vacancies.

```
method body how_many_vacancies(star: integer): integer
 in class City{
int number = 0;
o2 Hotel h;
for (h in self-->hotels where h-->free_rooms > 0
     && h-->starts == star)
         number += h-->free_rooms;
return number;
}
```

Clearly, O_2C was inspired by and retains the features of standard C.

O_2Look, the O_2 interface generator, supports "the display and manipulation of large, complex and multimedia objects on the screen [O2]." O_2Look is implemented in C++ on top of the Motif and the X Window System. There are four major features of O_2Look. The first is the ability to make "ready-made" presentations. Here, the user can view any part of the object and invoke any method. The presentation can be edited and then written back to the database. O_2Look supports cut/copy/paste operations on the structure of the object. Masks and resources can be used to customize presentations. A mask is a structure used to express which parts of an object should be visible and editable. A resource can be any graphic parameter of a widget: background color, fonts, layout, menu items, etc. Finally, O_2Look is extensible in the sense that new widgets can be defined by the programmer.

Performance Indicators

None are available

Platforms

O_2 runs on Sun workstations running UNIX, HP platforms running UNIX and the IBM RISC System 6000.

Concurrency

Soloviev [So] reports that O_2 supports "conventional transactions and performs concurrency control with two-phase locking protocol on files and pages. Concurrent access to objects is handled by WiSS [Wisconsin Storage System] used as the low-level Object manager Layer. However, an application cannot run more than one transaction at any particular time. The O_2C language supports only commit and abort commands for transactions. Restarts, checkpoints and nested transactions are not supported." O_2 does not support long transactions.

Distributivity

O_2 is based on a client server architecture. Client server communication is based on TCP/IP. A database may be distributed over the network as long as the clients have access to the disks on which the data is stored.

Price

The O_2 System for educational and research institutions costs $3,500 for a license for 1 to 7 workstations and $5,000 for 8 to 15 workstations. Each additional 15 workstations is $5,000. The standard price is $12,000 for the single user and $24,000 for a multiple user license. The O_2 Tool set is $12,000 for a single user and $8,000 for a multiple user license. Maintenance is 15% of list prices.

4.9 – Iris

Iris is a research prototype database management system under development at Hewlett-Packard Laboratories. It is intended to "meet the needs of new and emerging database applications, such as office automation and knowledge-based systems, engineering test and measurement, and hardware and software design [Fi]." Although Iris qualifies an as object-oriented database management system in the sense that it provides object identity and encapsulation, it relies on the translation of queries into relational algebra commands and the Iris Storage Manager is a "conventional relational storage subsystem ... similar to the Relational Storage System (RSS) in System R [Fi]."

Representational Ability

Iris supports multiple inheritance with some interesting options. The foundations of Iris may be found in the semantic data model known as DAPLEX and the Taxis language. DAPLEX is intended to facilitate database design by providing a rich set of object-oriented tools and a modest query language. The Iris type structure supports inheritance of attributes (or instance variables) and methods from parent types. Two sibling types may be declared to be disjoint in which case they may not share any instances. They may be declared to be overlapping if they are to share instances. One restriction results from two type declared to be disjoint; they may not have a common subtype since instances of the subtype would of course be shared instances of the parent types.

A property (method or attribute) may be generic in the sense that it may be defined for several types with the same name (but different definitions). Name conflict is resolved by inheriting the property from the most specific type (ancestor). "If a most specific type cannot be determined, user-specified rules for property selection will apply [Fi]."

Schema Evolution

Iris' schema evolution facilities are very primitive. A type may only be deleted if it has no descendants and no instances. New types can be added but a supertype/subtype relationship cannot be created for existing types.

Versioning

Iris supports versioning via Iris commands such as checkin, checkout, lock and unlock. These commands permit the creation and manipulation of versions and the sharing of versions among users. For example, we might create a version the object modlv1 with:

```
Checkout modlvl as modlv2;
```

The object modlv2 may be modified and returned to the database with:

```
Checkin modlv2;
```

The locks prevent other users from making versions of an object. For example,

```
Checkout modlv1 Key modkey1 as modlv2;
```

This command creates a version of modlv1 and prevents other users from the same act.

Query Facility

The Iris data model supports operations that are essentially functions in the sense that they always a result (value or object). For example, the specification of an operation might be:

```
        CREATE FUNCTION marriage(Person p) -> (Person spouse, Charstring date)
FORWARD;
```

The reserved word FORWARD means that function implementation will be provided later. The function marriage requires a single parameter from the class person and returns an object from the class person and a character string. The function might be invoked with:

```
Select s, d for each Person s, Charstring d
 Where <s, d> = marriage(bob);
```

where bob is the identifier of a person object. Strangely, a function may be stored as a table which in the above example would contain three columns for the person, spouse and date. Traditional functions are compiled into a relational algebra expression. A typical function of this type might be:

```
DEFINE manager(Employee e) = FIND Employee m
WHERE m = department-manager(department(e))
```

This query returns the manager of an employee object e by first returning the employee object's department which returns the department object's manager.

Iris also provides support for aggregate functions and the bag type. For example, the following query computes the average salary of all employees:

```
Select y For Each Real y
Where y = Average( Select Salary(x) For Each Employee x);
```

The nested select returns a bag of employee objects whose salary values are used as input to the average aggregate function.

Interfaces

Interfaces to Iris are via the library of C subroutines that define the Iris Object manager interface. High level languages that support calls to C routines can use the library as an interface to the database.

One interface provided by Iris is Object SQL, OSQL. Examples of OSQL can be found in the section on the query facility. The basic difference between OSQL and traditional SQL is the addition of the iterator "for each". OSQL may be used for interactive access to the database and has been embedded in Common Lisp via macro extension.

Iris also provides a graphical editor that allows the user to browse and update the Iris database. Using the graphical editor, we may display the structure of the database as well as the functions defined for the types and the instances of the type. Sample pictures from the graphical editor may be found in [Fi2].

Iris provides a tightly integrated interface via Persistent-CLOS (PCLOS). "PCLOS allows programmers to make objects transient or persistent by sending messages to them. ... Transient objects [from PCLOS] are not known to Iris [Fi2]."

Performance Indicators

Not available

Platforms

One could assume that since Iris is a prototype developed at the Hewlett-Packard Research, it runs on one or more HP platforms. Specific information regarding platforms is not available.

Concurrency

The ability of Iris to manage long transactions and versions is discussed in the section on versioning. Information specific to concurrency was not available.

Distributivity

Information not available.

Price

Since Iris is a research prototype, the issue of price is not applicable.

4.10 – Postgres

The Postgres database management system is an object-oriented extension of the Ingres relational database management system developed by a research team, headed by Michael Stonebraker, at the University of California, Berkeley. Postgres purports to not only provide object management to databases but to also provide knowledge management [St]. Object management is concerned with the problem of storing and manipulating nontraditional datatypes such as bitmaps, icons, text and polygons. Knowledge management, on the other hand, "entails the ability to store and enforce a collection of rules that are part of the semantics of the application [St]." Postgres is an instance of an object-oriented database whose query language, POSTQUEL, is an extension of the relational query language QUEL. QUEL is the query language supported by Ingres. In addition to providing access to Postgres via POSTQUEL, users may access Postgres data via CLOS (Common Lisp Object System). Access via C++ should be available soon. Postgres also supports a "fast path" interface that we shall describe in the section on interfaces.

Representational Ability

Postgres supports traditional object-oriented paradigm with classes, encapsulation and inheritance. Classes are very simply defined by specifying the class name with instance variable names and their types. Subclasses are defined with additional instance variables and specification of parent classes from which to inherit behavior and attributes. Postgres supports multiple inheritance with the caveat that a subclass cannot be created if it causes name conflict resolution. In the case of name conflict, Postgres refuses to create the class. Postgres classifies classes into one of three categories. A real or base class is one in which the instances are stored in the database. A class can be a derived class (or view or virtual class) whose "instances" are not actually stored but are available when needed. This, of course, corresponds to the notion of a view in the relational model. The third category is the case where a class can be a version of another class.

Postgres provides support for three kinds of types. A base type which extends the machine types such as integer, float, character string, etc. Postgres provides an "Abstract data type" facility which supports the users' ability to construct new base types. The user must provide a function to map character strings to and from the base type. To illustrate these concepts, we appeal to the examples provided in Stonebraker's paper [St]:

```
Create EMP (Name=c12, Salary=float, Age=int)
```

In this example, we have the classical definition of a class (type) with instance variables having standard data types. We can now define a subclass with

```
Create SALESMAN (Quota=float) inherits EMP
```

The class SALESMAN is a subclass of EMP and as such, inherits the three instance variables from EMP as well as any functions defined for EMP. We can now define a class which illustrates the ability to define an Abstract Data Type (ADT).

```
        Create DEPT (Dname=c10, Manager=c12, Floorspace=polygon,
             Mailstop=point)
```

The types polygon and point are base types defined by the user.

A second kind of type provided by Postgres is an array type whose entries are base types. The entries in an array type are indexed by the natural numbers.

Finally, composite types permit the construction of complex objects. A composite type accepts values which are instances of other classes. For example, we might have defined the class EMP as:

```
        Create EMP (Name=c12, Salary=float[12], Age=int,
        Manager=EMP, Coworkers=EMP)
```

The instance variable Salary is now an array which permits individual values for each of the 12 months. The attributes Manager and Coworkers may reference zero or more instances of objects from the class EMP. In this case, the definition of a class automatically provides a new type whose values are members of the class.

Postgres also supports an explicit composite type which can hold objects from a diverse collection of classes. For example, suppose the class EMP required an attribute Hobbies to reflect the hobbies enjoyed by the individual. The values for the attribute Hobbies may be collections of objects from the class Softball or Water_Skiing or Hiking, etc. We can actually modify the schema of EMP with

```
        Add to EMP (Hobbies=set)
```

Postgres supports a feature of its query language that permits the user to pose queries on composite types by using the nested dot notation. The nested dot notation has also been encountered in Gemstone, ITASCA, ORION and O_2 to name a few. For example, to retrieve the age of Joe's manager in the EMP class, we can pose the following query:

```
        retrieve (EMP.Manager.Age) where EMP.Name = "Joe"
```

This query retrieves an object from the class EMP having the Name attribute value Joe; it then references an object in the EMP class via the attribute Manager. From this object, the value of the Age attribute is returned.

One of the features specific to Postgres is its support and use of functions. Postgres supports three different kinds of functions: C functions, operators and POSTQUEL functions. C functions are analogous to methods in the traditional object-oriented system. C functions accept base types or composite types as arguments. For example, we might define a C function for the class DEPT which returns the size of the floorspace of a department object. The name of the C function is floorspace and may be invoked as follows:

```
        retrieve(DEPT.Dname) where (DEPT.floorspace) > 500
```

Here, the C function "floorspace" is associated with the class DEPT and applies to individual instances of that class. A C function can also be defined for the class as a whole, i.e. a class method. As with methods, C functions are inherited by any descendant class of a class.

An operator is a function whose purpose is to facilitate indexing in processing queries. An operator has one or two arguments just as a traditional operator in a classical programming language. For example, we might define an operator AGT (area greater than) that compares two floorspace values from the class DEPT. For example, consider

```
retrieve (DEPT.Dname)
where DEPT.Floorspace AGT "(0,0), (1,1), (0,2)"
```

Here, the floorspace value of an object in the class DEPT is compared to a literal floorspace value.

Finally, we may collect several POSTQUEL commands and combine them into a single function. Stonebraker [St] observes that "any collection of commands in the POSTQUEL language can be packaged together and defined as a function." For example, we might want to define a function which returns objects of the class EMP having a salary exceeding $50,000.

```
define function high-pay returns EMP as
retrieve (EMP>all) where EMP.Salary > 50000
```

This type of function is not unlike a view in the traditional relational model.

POSTGRES has a rather unique rules system that "... could perform all of the following functions: view management, triggers, integrity constraints, referential integrity, protection and version control [St]." The syntax of a rule is:

```
ON event (TO) object WHERE
POSTQUEL-qualification
THEN DO [instead]
POSTQUEL-command(s)
```

Here, an event is a retrieve or a replace or a delete or an append or a new or an old. An object is either the name of a class or an attribute (Class.Column). The appearance of the optional instead means that the action is to be performed instead of the action that precipitated the rule. If instead is missing, then the action is to be performed with the action that precipitated the rule. For example, suppose that Fred's salary is to always match that of Joe's. We would like a rule that modifies Fred's salary whenever Joe's salary is modified [St]. This is accomplished with the following rule:

```
ON new EMP.Salary WHERE
EMP.Name = "Joe"
THEN DO replace
E (salary = new salary)
from E in EMP
where E.Name = "Fred"
```

Schema Evolution

Stonebraker [St] does not explicitly discuss POSTGRES's support for modifying the schema. There are some glimpses in the paper that some changes are possible. It seems that we can add attributes dynamically and the modifications apply to objects that have been created for the class. In the discussion on implementation, we find that "POSTGRES assumes that data types, operators and functions can be added and subtracted dynamically, that is, while the system is operating [St]." There is no mention of the ability to add classes within the lattice. It is assumed that leaf classes can always be added.

Versioning

POSTGRES supports an explicit command to create a version of a class. For example, to create a new version of the class EMP, we might have:

```
create version New_EMP from EMP
```

This command automatically creates two additional classes; EMP_MINUS and EMP_PLUS, and adds a collection of rules to the set of rules for the database. The class EMP_MINUS contains references to any object which is to be deleted from the version. EMP_PLUS contains references to new instances added to the version as well as modifications to instances of EMP. The rules that are included with the creation of the version include:

```
ON retrieve TO New_EMP
THEN DO instead
retrieve (EMP_PLUS.all)
```

Additional rules are created for deletion, replace and append. There does not seem to be a facility for long transactions or a check-in/check-out option. Versioning seems to apply to the class as a whole.

Query Facility

POSTQUEL is an extension of the relational query language QUEL which was the query facility for the relational database management system INGRES. POSTQUEL actually contains QUEL as a subset. POSTQUEL has extended QUEL by adding support for nested queries, transitive closures and inheritance. The transitive closure supports recursive retrievals. For example, suppose that we have a class representing a directed graph given by:

```
create Arc( From int, To int )
```

If (1,2) is an object in the class Arc, then there is an arc from node 1 to node 2. We can now request all nodes reachable for node 1, say, with:

```
retrieve* into Path( Arc.To ) from p in Path
where Arc.From = 1 or Arc.From = p.To
```

The * indicates that the query should continue to execute until no new objects can be found that satisfy the condition.

Additional examples of POSTQUEL appeared in the section on representational ability.

Interfaces

An independent research group has developed a CLOS (Common LISP Object System) interface to POSTGRES. Stonebraker reports that a C++ interface is under development.

Platforms

POSTGRES runs on Sun-3, Sun-4, DECstation and Sequent Symmetry machines.

Concurrency

POSTGRES supports concurrency to a limited extent. Rather than have one process executing, POSTGRES runs an individual process for each user. The intent was to acquire an operational product as quickly as possible.

Distributivity

It does not appear that POSTGRES supports a distributive environment.

Price

POSTGRES is free and may be acquired via Internet or via tape.

5 – Bibliography

[Ah] Shamin Ahmed, Albert Wong, Duvvuru Sriram and Robert Logcher, "A Comparison of Object-Oriented Database Management Systems for Engineering Applications," Intelligent Engineering Systems Laboratory, Massachusetts Institute of Technology, May, 1991

[An] Timothy Andrews and Craig Harris, "Combining Language and Database Advances in an Object-Oriented Environment", Proceedings of the OOPSLA, 1987

[At] Malcolm Atkinson, et.al., "The Object-Oriented Database System Manifesto", Deductive and Object-Oriented Databases, Elsevier Science Publishers B.V., North Holland, 1990

[Ba] J. Banerjee, et. al., "Data Model Issues for Object-Oriented Applications," ACM Transactions on Office Information Systems, Vol.5, No. 1, 1987

[Bo] P. Borras, J.C. Mamou, D. Plateau, B. Poyet and D. Tallot, "Building User Interfaces for Database Applications: The O_2 Experience", SIGMOD RECORD, Vol. 21, No. 1, march , 1992

[Bu] Paul Butterworth, Allen Otis and Jacob Stein, "The Gemstone Object Database Management System", Communications of the ACM, Vol. 34, No. 10, October, 1991, pages 64 - 77

[Ca] R.G.G. Cattell, An Introduction to the Communications of The ACM issue on Next-Generation Database Systems, October, 1991, Vol. 34, No. 10

[De] O. Deux, "The O_2 System", Communications of the ACM, Oct. 1991, Vol.34, No. 10, pages 35 - 48.

[Ei] David Eichmann, "A Survey of Types, Abstraction and Schema Evolution in Object-Oriented Systems", Unpublished

[Fi] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan and M.C. Shan, "Iris: An Object-Oriented Database Management System," ACM Transactions on Office Information Systems, Vol. 5, No. 1, January, 1987, pages 48 to 69.

[Fi2] Alponso F. Cardenas and Dennis McLeod, Research Foundations in Object-Oriented And Semantic Database Systems, Prentice Hall, 1990.

[Ge] Gemstone Product Overview, Servio Corporation, February, 1992

[HK] Richard Hull and Roger King, "Semantic Database Modeling: Survey, Applications and Research Issues", ACM Computing Surveys, Vol. 19, Number 3, Sept., 1987

[Hu] John G. Hughes, Object-Oriented Databases, 1991, Prentice Hall International (UK) ltd.

[It] ITASCA: Distributed Object-Oriented Database Management System, Technical Summary

[KC] Setrag N. Khoshafian and George P. Copeland, "Object Identity", Proceedings of Object Oriented Programming Systems, Languages and Applications-86, ACM, 1986

[Ki] Won Kim, Introduction to Object-Oriented Databases, 1990, The MIT Press

[La] Charles Lamb, Gordon Landis, Jack Orenstein and Dan Weinreb, "The ObjectStore Database System," Communications of the ACM, October, 1991, Vol. 34, No. 10, pages 50 to 63

[Le] Christopher Lecluse, Philippe Richard and Fernando Velez, "O_2, an Object-Oriented Data Model", Proceedings of SIGMOD, 1988

[Ma] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS", Proceedings of the 1st International Conference on Expert Database Systems, 1985

[Ne] Michael Nelson, "An Object Oriented Tower of Babel", OOPS Messenger, July, 1991, Vol. 2, No. 3

[Ob1] Object Databases, Vendor's Product Announcement and Brief

[Ob2] ObjectStore, Promotional Literature and Technical Overview provided by the vendor, Object Design

[Ob3] Objectivity, Vendor's Product data Sheet and System Overview

[Or] The ORION Papers: Research Reports from MCC's Object-Oriented and Distributed Systems Laboratory

[O2] Vendor supplied product description and Technical Overview

[O2U] The O_2 User's Guide, Version 2, March, 1992

[Sn] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", Proceedings of OOPSLA-86, pages 38-45, 1986

[So] V. Soloviev, "An Overview of Three Commercial Object-Oriented DBMSs: ONTOS, ObjectStore and O_2", SIGMOD RECORD, Vol. 21, No. 1, march, 1992

[St] Michael Stonebraker and Greg Kemnitz, "The Postgres Next Generation Database Management System," Communications of the ACM, October, 1992, Vol. 34, No. 10, pgs 78 - 92

[Ve1] VERSANT in Brief, Documentation provided by Versant Object Technology

[Ve2] VERSANT, System Reference Manual for Sun Workstations, September, 1991

[Zd] Stanley B. Zdonik and David Maier, Readings in Object-Oriented Database Management Systems, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990