Report on the

OBJECT-ORIENTED DATABASE WORKSHOP:

Implementation Aspects

Held in conjunction with the

Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87) Conference

October 5, 1987
Orlando, Florida

Satish M. Thatte

Workshop Chairman

Artificial Intelligence Laboratory
Texas Instruments Incorporated
P.O. Box 655474, M/S 238
Dallas, TX 75265
CSNet: Thatte@ti-csl

1 Introduction

Object-oriented database systems combine the strengths of object-oriented programming systems and data models, with those of database systems. This half-day workshop focused on issues such as object persistence, persistent object storage servers, object sharing, transactions on objects, query optimization, and performance issues (buffering, prefetching, clustering, etc.)

The workshop panel consisted of eight members: Craig Damon (Ontologic), Sandra Heiler (Computer Corporation of America), David Maier (Oregon Graduate Center), Patrick O'Brien (Digital Equipment Corporation), Lawrence Rowe (University of California at Berkeley), Alfred Spector (Carnegie-Mellon University), David Wells (Texas Instruments), and Stanley Zdonik (Brown University). In the first 90 minutes, each panel member presented his/her position. This was followed by questions from the workshop participants and discussion.

To encourage vigorous interactions and exchange of ideas between the participants, the

workshop was limited to participants based on the submission of an abstract describing their work related to implementation issues of object-oriented database systems. The workshop announcement drew a very enthusiastic response. 35 single authors and 18 co-authors submitted abstracts. 57 participants (including panel members and walk-in participants) attended the workshop.

This report is organized into two major sections. Section 2 presents position statements of the panel members. Section 3 transcribes questions asked by the workshop participants to panel members and their answers. Section 4 concludes the report.

2 Panel position statements

VBase object-oriented database system

Craig Damon, Ontologic

Vbase is the object-oriented database system being built by Ontologic. Much of its linguistic heritage is drawn from the language CLU developed at MIT. It is a compile time type safe system with full inheritance. Vbase supports persistent, shared and local objects. The system is written in the two Vbase languages, TDL, the type definition or schema language, and COP, an object- oriented extended C.

One feature of the Vbase system which has proven a great boon to the implementation is the clean distinction between abstract types and storage types. Most object systems provide only a single storage type for all instances, or perhaps a single storage class for all instances of a single abstract type. In Vbase, an abstract type defines a set of abstract behaviors (operations, properties, attributes, relationships, etc.) which are the sole external interface to instances of the type. The type manager implements these behaviors internally by storing and manipulating its instances' representation. It is the storage type which handles management of the representation's storage, including dereferencing, object faulting, clustering and object persistence. These behaviors are associated with "storage class" rather than with "abstract class."

An abstract type provides a default storage class which implements these behaviors. A type might limit the set of storage managers that it supports, either for performance reasons (as in the case of integers) or semantic reasons (as in the case of certain system types which must reside in shared memory). Typically, however, a type will allow a wide range of storage classes. This vastly simplifies adding a new storage class for use by existing abstract types. We added a process local (heap based) storage class in a few days, making non-persistent instances of the existing abstract types possible.

Raising the storage problem to being a semantically visible portion of the system has another clear advantage for clustering. The storage managers all support a series of clustering levels. The normal arrangement is to support chunks (contiguous storage), segments (a clustering of chunks) and areas (a clustering of segments); however any given storage manager is free to support any number of levels of clustering. For the default persistent storage type, a segment is the unit of transfer to and from disk and an area maps to a disk

partition or file. As an alternative, process local objects have only two levels of clustering: chunks and segments, where segments are built along operating system paging boundaries in a virtual memory system.

Since all storage managers support the same basic model of clustering, it becomes a much simpler problem to export a strong clustering pragma language to the type implementor. In Vbase, objects may be clustered at any clustering level, along semantic relationships. Typically, all of the separate storage pieces required to implement an object are clustered within the same chunk. This chunk is then clustered within the same segment as the chunks implementing another related object. For example, in a CAD application, all of the sub pieces of an assembly might be clustered within the same segment, thus requiring only a single disk transfer when accessing the assembly. Our experience in tuning applications to take advantage of the clustering capabilities of the system has been very favorable. We have reduced the disk activity of one application by more than a factor of five with only a change to the clustering actions.

Mapping Objects from Disk to Memory

David Maier, Oregon Graduate Center

The desired properties for objects stored on disk and objects in use by an application program are quite different. A key desirable feature for disk storage is flexibility. We want objects and types to be free to change, we want to capture multiple aspects of an object, and we may want to support free-format portions of an object state. Further, we often want to access objects on disk by associatively accessing their states. Within program memory, however, we are willing to trade flexibility for faster access. Usually, a particular program is only interested in a certain aspect of an object, and thus, only part of the object's state. We want object access in program memory to be competitive with access to record and array structures – being able to proceed by addressing rather than by searching. We are investigating ways to efficiently map flexible, variable-size objects on disk to more structured fragments of object state in program memory.

The OBJFADS Shared Object Hierarchy

Lawrence Rowe, University of California, Berkeley

I strongly believe that stand alone object-oriented databases do not solve very many real-life problems. You must solve problems that involve business data. When I talk to people that are building real world applications, they are not interested in yet another database system.

OBJFADS is an object-oriented programming environment for POSTGRES, a next-generation database system designed to support engineering/scientific and expert system applications. OBJFADS has a shared object hierarchy (i.e., a portion of the object hierarchy is shared with other users), a direct manipulation interface, extensible interface

abstractions (e.g., active forms with user-defined field types), and integrated application generators (e.g., query/update interface generators). POSTGRES is an extensible relational DBMS (e.g., users can define new attribute data types and access methods) that provides support for complex objects, procedures (i.e., data of type "procedure"), active databases (e.g., alerters, triggers, and rules), precomputed values (e.g., procedures and rules), and historical data (including versions and snapshots).

The OBJFADS shared object hierarchy is stored in a POSTGRES database. The major problem with this approach to sharing is efficient implementation of object referencing and updating. Object referencing time can be reduced by maintaining an object cache in the application program, by precomputing the main memory object representation, and by pre-fetching related objects.

The object cache reduces the time to fetch an object that is referenced frequently. POSTGRES precomputed procedures will be used to reduce the overhead required the first time an object is fetched from the database. Complex objects are typically represented by tuples in several relations. To fetch an object, these relations must be "joined" and the data attributes must be coerced into the appropriate application data types. A precomputed procedure can maintain a copy of the main memory object representation in the database. Precomputation will reduce the time required to fetch the object while, at the same time, maintaining the relational representation in the database. Lastly, prefetching will be used to reduce the probability that an object will have to be fetched from the database when it is first referenced. Applications typically reference complex objects composed of several related objects. Prefetching all objects that make up the complex object reduces the database fetches.

Because the DBMS has user defined types, and because we are going from a database type system to a type system in the program language, a general type mapping facility is needed. For existing database systems, the types are fixed. Therefore, existing database systems pretty much wire the mapping into the run-time system of a programming language. In contrast, in POSTGRES, objects can be shared and they can reference local objects. Therefore, when a shared object is saved in the database system, that object needs to be "unbound" from the run-time system, and a description of how to recreate it needs to be stored in the database. Similarly, when the object is retrieved from the database into memory, that object needs to be recreated from its description stored in the database. We ran into lot of problems in this process when we dealt with objects such as windows and I/O ports. For example, when an object is retrieved from the database, it has windows defined in the X window server. It must be able to recreate those windows from a description stored in the database on how to generate them.

Support for composite objects is absolutely needed. Won Kim's paper on composite object in the OOPSLA '87 proceedings looks real interesting. There are two ways to support composite objects. If you do records and pointers, you get into the prefetching and clustering "racket." The second thing you can do is to use relations with precomputed procedures. The neat thing about precomputed procedures, for at least the queries they express, is that they do perfect caching. It makes query optimization a lot easier.

The Utility of a Uniform Distributed Transaction Facility for

Supporting Object-Oriented Databases

Alfred Spector, Carnegie Mellon University

Databases of all types, be they as simple as file systems or as complex as the latest extensible database management systems, can benefit from a carefully implemented distributed transaction facility. Such a facility should, at minimum, support inter-transaction synchronization; recovery after transaction abort, and node, server, and media failures; commit management; and location-transparent calls on objects. I argue that such facilities can have easy to use interfaces and be highly efficient. As an example, I will briefly sketch the interface to the Camelot Distributed Transaction Facility, which my group has built at Carnegie Mellon. Camelot is intended for production use in a number of application environments, including on-line transaction processing and special purpose databases. Camelot Release 0.7(31)-aleph presently runs on the Unix-compatible, Mach operating system on IBM RT PCs, DEC Vaxes, and Sun 3's in the DARPA internet environment.

Camelot provides coherent layers whereby we permit users to install subsystems. We permit people to issue remote procedure calls or methods on objects. We support all the commit protocols that you can conceive of. We handle recovery and provide security facilities. We have Mach, which is a Unix compatible distributed operating system that CMU is building and distributing. On top of Mach there are communication components, TCP/IP layer, and a message model with an RPC stub compiler. On top of that we have Camelot transaction facility, and then finally we can layer servers. Jeannette Wing at CMU is overseeing an effort to build a C++ derivative that has persistent objects that utilize recovery and transaction support in Camelot.

Camelot is designed to support implementation of other object bases. We have plans underway to support a relational database system as well. At the kernel we have a process which restarts the system after crashes. We do our own disk management for objects. We do write-ahead logging based upon information from the data servers as to what they want logged, so we can take into account their semantic knowledge. Our programming libraries will do a pretty good job of deciding the minimum amount of logging to do for value logging. For operation logging, which we will support shortly, the programmer will have to do more. Communication manager makes RPC work, tracks what transaction is spread over the network so we can handle the commitment of transactions. The transaction manager supports Eliot Moss's nested transaction model, supports a number of other kinds of transaction, like commit protocols including a non-blocking commit protocol, which reduces the likelihood that data remains blocked due to a coordinator crash, and the recovery manager uses the log to get you out of abort problems, transaction, aborts, node failures or immediate failures.

Issues in object-oriented database implementation

Stanley B. Zdonik, Brown University

I will describe some general principles for implementing object-oriented databases and persistent object systems. I will briefly touch on requirements for object storage, concurrency control, and distribution. These ideas will be illustrated with experience gained from the implementation of our object-server named ObServer and our object-oriented database system, named ENCORE. Encore provides basic database capabilities, such as types, methods, properties, inheritance, that you would expect in an object-oriented database. Below Encore lies ObServer. It is typeless and knows nothing about high-level semantic models used by Encore. ObServer essentially provides a mapping between object identifiers and storage chunks. Also it has facilities for handling concurrency, clustering, recovery, and other aspects of storage management. ObServer is running, and in fact, is being used by Steve Rice in the implementation of his Garden programming environment. Garden is a graphical programming environment that allows programs to be developed by manipulating eye contact and drawing pictures. There are many things in ObServer that were responses to the needs of Garden environment.

Garden as our first client, drove us to choose a typeless model in ObServer. Garden people had already written a lot of code, and wanted to have a type model of their own, and wanted to be able to support that model without rewriting some of the code. A consequence of the fact that ObServer is typeless is that it is easier to build other kinds of models, for example, ENCORE, on top of it. There are some questions about the current implementation. For example, we were wondering whether it would be better to have a model in ObServer that allows the identification of object pointers rather than storing only arbitrary storage chunks. This would allow us, for example, to do garbage collection on ObServer. Since we are doing some concurrency control in ObServer, some notion of versions at this level may be useful to handle conflicts.

ObServer supplies more than just the standard locking modes. We do supply read and write locks, and if you should use those in a two-phase way, then you get well-known serialized transactions. People occasionally want transactions which are non-serial to achieve high performance. We provide *notify* and *write-keep* locks for this purpose.

ObServer has a notion of segments. A segment is the unit of clustering. It is also the unit of read operation from ObServer. ObServer will return all the objects in its segment. Segments can be of any arbitrary size. UNIX does not do a very good job of providing segments that are contiguous on disk. Therefore, we actually built an experimental file system that guarantees logically contiguous segments of arbitrary size on disk. Segment loading is accomplished by simple heuristics implemented at Encore level, like store all objects of given Encore type together, store all objects with a common value or property together. If these simple heuristics are not adequate for an application, then it is possible at the application level to decide how to load the segments.

I claim that for applications, queries and query optimization are not that important anymore. You will still do them, but you will not do them as much. This is my view of how an engineer works: he comes in the morning, he switches his machine on, he issues a query, and he loads up his working set. Then he follows relationships for the rest of the day. That means, graph traversal is really what he does a lot of. Therefore, we need not focus so much on query optimization; instead we have to solve this graph traversal

problem. This means that you have to reduce the number of object faults and do a good job of object clustering.

Object-Oriented Engineering Information System to support VHSIC design

Sandra Heiler, Computer Corporation of America

We are currently in the process of designing and developing an object management system, which will be the major component in an object-oriented Engineering Information System. The goal of this system is to support Very High Speed Integrated Circuit (VHSIC) design by providing services to integrate software and hardware tools from different vendors, to manage engineering data, and to provide support for the management and control of engineering processes.

The development effort will provide a set of specifications and core services to support tool integration and management without requiring the various installations that use such a system either to re-implement the whole system or to change the way they do business. This means that the resulting system must be tailorable to existing configurations of hardware, software tools, and data repositories and must enforce the engineering and management policies of each installation.

The object manager has much in common with an Object-Oriented DBMS (OODBMS). It extends the functions of such a system, though, in several ways: First, it must manage non-database objects, i.e., objects that have no stored extents (for example, dynamic objects such as the results of simulations that are recomputed each time they are accessed).

Second, the object manager must support arbitrary operations on these objects, in addition to supporting traditional database operations. For example, the object manager must support the running of engineering tools on design objects. Engineering operations may span days, expanding OODBMS issues of object persistence, concurrency control, and recovery.

Third, the object manager must use supporting services provided by a collection of heterogeneous hardware and software. For example, the object manager is not responsible, itself, for the storage and manipulation of database objects. Instead, it relies on a set of underlying DBMSs to actually perform storage and manipulation operations. These underlying facilities will range from powerful general-purpose object-oriented DBMSs to very weak or specialized proprietary products.

In this environment, method management becomes a critical issue. The issue arises because location and invocation of a method often requires dynamic decisions based on such information as the availability of system resources (optimization issues) and object state information. For example, various forms of the same method may be executable in different environments, the desired method code may be located remote from the host on which it must be executed, or a particular object version may require an older version of the method code. Optimization in a distributed, heterogeneous environment may involve decisions including method run-time requirements, node availability, resource utilization,

and data movement considerations (e.g., a decision to move method code to the object vs. moving method parameters to the code).

Object-Oriented Database Extension to Trellis

Patrick O'Brien, Digital Equipment Corporation

The Object-Based Systems Group is designing an object-oriented database extension to Trellis, an object-oriented language and programming environment developed at DEC. The primary purpose of the database, which we call an "object-repository," is to provide shared access to persistent objects in a multi-user environment. It will also provide the usual database amenities such as concurrency control, recovery, and authorization. The database is intended for applications, such as engineering data management, which have complex data structuring requirements and special data accessing needs.

Our objective is to achieve a tight integration of database management concepts within the object-oriented style of Trellis/Owl. This contrasts with traditional language/database interfaces that rely upon a set of function calls or a separate data manipulation language which has little or no interaction with other language features. With the traditional approach, the database complicates the programmer's view by introducing a second level of storage with different data modeling constructs. As well, some objects are temporary, others are persistent and shareable, and decisions must be made about when to update objects in the database. We would like to make the interface between the programming language and the database as transparent as possible (i.e. maintain an illusion of a one level store).

Our database extension to Trellis/Owl is presented to the programmer as new types added to the library. We use existing facilities for declaration of types and variables, along with existing control structures for iteration and exception handling. This allows programs to access and manipulate database objects in the manner as all other objects. The repository and the Trellis/Owl language support the same data structuring capabilities so there is no impedance mismatch.

Concurrency and recovery are also important issues. One of the main reasons for having object repositories is to allow applications to share information. Conventional database concurrency mechanisms, however, are too restrictive for the design environments that we want to support. For example, simple transactions based on locking can provide for atomic update and recovery, but they tend to force a much more serial schedule than is logically necessary. Furthermore, they prevent a transaction from seeing the intermediate results of another transaction. We do not believe that we can define the semantics of sharing for all applications and provide one synchronization mechanism that will suffice for all applications. Our general philosophy is to provide the proper level of primitives so that applications built on top of the system can present the transaction mechanism that best suits their environment. Thus, the repository will provide low level mechanisms for coordinating reliable updates to objects, and these mechanisms can be tailored and extended for a particular application.

How object-oriented databases are different from relational databases

David Wells, Texas Instruments

I see three major differences between OODBs and relational systems: complex, highly interconnected objects; access driven by the data rather than the program; and very complex translation routines. I will discuss what I see as the implications of these differences.

Because inter-object references are stored directly in an OODB, an object's identity must be invariant over changes to its state, and time must be a qualifier of identity to allow references to specific states of an object or to "most recent." Object boundary is a nasty problem. A user should be able to save objects without worrying about object boundaries. Object closure is not an acceptable definition of boundary, due to its potential size in Lisp-like environments. Also, do we really need to *store* an object's entire value? If we are pointing to a buffer, should we save that buffer, or should we throw it away and recreate it when required? "Local persistence" further complicates the issue. STANDARD-OUTPUT should be a persistent object because it may be referenced by other persistent objects. However, when a persistent object moves from machine to machine, it would reasonably be expected to use the local standard output.

OODBs have a navigational feel; I go from one object to another, and I look inside an object to determine reasonable places to go next. This is unlike a typical relational query, where you can look at the program and see where you are going. Since access hints are stored within the OODBs, we should be able to do much more in the way of intelligent storage management, concurrency control, caching, and prefetching than in relational systems. At the same time, concurrency control is complicated because in the limit we can often reach the entire object space, thereby precluding the use of a priori locking. Care must be taken to avoid overly restrictive lock sets.

For OODB translation routines are going to be more complex, implying that the system should be divided into a semantic module and a storage module to allow their reuse. Translation requires a lot of relocation in addition to pointers and virtual machine references. The handling of global references in local environments is an open issue. If you store a function in an OODB and bring it back on a different machine, can that function have free variables, and if so, how should they bind in the new environment? If we do bind to a local environment, does that mean that to store a function, you must guarantee that it can always load?

3 Discussion

There was a lively discussion on many issues, such as the role of parallel computer architectures for OODBs, concurrency control of complex objects, clustering and prefetching, garbage collection of persistent objects, query optimization, and operating system support for databases.

Michael Caruso, Innovative Systems: It seems to me that implicit in the presentation of some panel members is a very elementary "record-at-a-time" model of computation based on the Von Neumann architecture. But if you free yourselves from these assumptions you start to get some answers to questions on clustering and partitioning of objects, and graph traversal algorithms. I would suggest there are other alternatives to record-at-a-time architecture. With these alternatives you can think about encapsulating a time duration and can move the notion of iteration encapsulation down to a lower level in the architecture. It is a fundamental characteristic of database systems to encapsulate the iteration abstraction at a very low level in order to make the system perform well. We view this alternate computer architecture as a parallel/pipelined processor to allow a query to operate in parallel on elements of a collection. The idea of binding a message to an operation and binding a name to an operation can be factored in different ways. I can exploit the fact that certain information about that collection is structured in such a way that binding of the message is not necessarily done element at a time. The other thing we can do is to execute programs in a pipeline fashion. A program is not really interested in seeing information a whole object at a time. What it is interested in doing is to see a function at a time, for a collection of objects. You can think about alternative clusterings for objects, carving up objects, not horizontally, but vertically, and arrange structures in memory to optimize this vertical partitioning and graph traversal in parallel.

Eliot Moss, University of Massachusetts: I would like to say something back to Michael Caruso on this. My recollection is that his company has been looking primarily at statistical and managerial databases. The argument I might make, if I put on David Wells' hat for a minute, is that the kind of data David is looking is different; he is going to do more navigation, instead of parallel access. So, access patterns will be a property of data.

Michael Caruso, Innovative Systems: I am interested in complex structures and how they decompose. What I have is a set of objects, but for that set I will do navigational access.

Remark from the audience: All of us agree that we would like to have multiple interfaces to data. Some are "record-at-a-time," some are at a higher level. The higher level interfaces allow cleverer strategies within the object or server. I think that is the point you (Michael ?) were trying to make.

Frank Manola, GTE: Can we agree, or agree to disagree, on the fact that people want to use database systems for different kind of data. Engineers want to access data that might involve a more "object-at-a-time" approach, and people who do conventional database applications, like payroll, are going to access "set-at-a-time." The problem I see is that historically people have tried to use databases for everything. When we talk to people like General Dynamics, they are not interested in having two incompatible database systems, one for their engineering data and one for their business data. They are interested in having one database system in which they can store all data. Therefore, it seems to me that for some queries the employee file is going to be processed the same way as it always has been, namely set-at-a-time, in spite of the fact that one of the new things that it can store about employees is their photographs and other complicated information.

Can we look at implementation techniques that involve satisfying both set-oriented access and set-oriented query optimizations that we know how to do, as well as object-at-a-time unpredictable access patterns for new applications that have driven object-oriented database implementations recently.

I have now heard the CODASYL *area* concept reinvented for the third time in the last one and half days! People found that very difficult to use in doing storage clustering. I think it is a straight-forward idea, it is intuitively nice, and I have no objection to it. I just wonder, because the things being put in these clusters are called objects instead of records, all of a sudden how people are going to make those things work. I think we ought to keep in mind that a lot of people want to use database systems for corporate databases, not just to support engineering design tools.

David Wells, Texas Instruments: With respect to the notion of parallel queries and parallel processing, it seems to me that the correct solution to that problem is to allow a set to be an object or an object to have the value of a set. Once you have got the set of objects, it is up to the user to process them anyway he wants, and each processing thread then becomes independent with respect to the database. If each one of these threads is following essentially a depth-first traversal, then the clustering technique ought to be smart enough to recognize that as we go from one object to the next one.

We are looking at doing the clustering in the object server, which will collect statistics and will statistically determine what objects ought to be clustered. We suspect very strongly that clustering will have to be related to application. Essentially, an application would say "I'm an editor, or I'm a compiler, or I'm a CAD system," and then you would get different kinds of prefetching based on the application. We will collect statistics differently. We also suspect that we will go through a learning phase and a using phase, because statistics collecting is going to be a little bit expensive, and also for the object faulting mechanism, you want to know why an object has been fetched, as opposed to simply the fact that it has been fetched. So in our approach the storage system itself will collect those statistics, and we will get away from the notion that a user has to specify clustering (objects going in areas). From our garbage collection experience on Lisp machines, users most often do not tell you the right things and therefore you are better off not doing anything.

David Jordan, Bell Labs: I am talking from a user's perspective. It's a CAD/CAM type application. We have got a need to store a multi-level hierarchy of information representing a product design. Regarding multi-level hierarchies, I was wondering what work has been done in terms of locking, clustering, and access? For example, for locking, a given complex object consists of many sub-objects, and you would like to be able to essentially lock sub-trees without interference. If someone tries to lock a global complex object, you need to know that sub-objects have been locked and vice-versa. You do not want to go in and lock sub-objects if the global object has been locked. All the clustering implementations I have seen are just one level. It seems like we could use clustering both from a depth-first and breadth-first order depending on what type of access we anticipate. For global complex objects, we typically have applications that are going to access a particular sub-tree.

Lawrence Rowe, University of California-Berkeley: Read Jim Gray's paper on locking published back in 77-78. Several people have looked at hierarchical locking to support composite or complex objects, where you can set the lock on the large object and then whenever you come in and lock you must go through the lock hierarchy in the correct order. In terms of clustering, whether you chose to enforce one or two policies, or whether you have a system that learns based on usage or on some other way of specifying sets of records that should be co-located, is one of the big controversial issues. If there were a clear answer to clustering, I am not sure we would be having this panel!

Stanley Zdonik, Brown University: As far as the first question on locking complex objects is concerned, what you mean by complex objects varies from application to application, in fact from operation to operation. As somebody else on the panel mentioned, it is not correct to necessarily lock all its pieces whenever you touch the top level of a complex object. For each operation, you need to build in the knowledge (either in the code or somehow else declaratively), about what pieces need to be locked. I do not think there is a general solution that has to go with the application code. In our view, clusters are just buckets. You can throw whatever you want in them. As far as clustering hierarchies are concerned, I do not understand your (Jordan's) question, because we do not have a one-level view of clustering. You can throw the entire complex object into a cluster if that is what your wish.

David Wells, Texas Instruments: I want to make a point with respect to hierarchical locking. We do not really have a strict hierarchy. We have a directed graph, and therefore, it is not clear that a strict hierarchical locking policy is going to work, because there is more than one way to get to an object. You have to know all of the objects within a closure of an object to determine whether the closures overlap. That is difficult for two reasons, one is that the closures often are very large, and that means keeping track of a large amount of information. When you try to determine whether two closures overlap so that you can allocate a lock, you have to look at an awful lot of objects, which may not even be on the same server. If it is a distributed system, determining the closure overlap is going to take you all over the place. It is not clear that the method will work in practice.

Roger King, University of Colorado: Since you (Stanley Zdonik) have a separation between the ObServer and layers above it, do you find that a lack of semantics at the lower level cause problems, such as storing differentials ?

Stanley Zdonik, Brown University: Yes, they do cause problems. There are clearly tradeoffs of having no semantics at the ObServer level; you cannot do much there that you might want to do (such as garbage collection). That is why I hinted at the idea of putting some semantics, such as pointers, storage layout, and versions. I view the boundary between ObServer and Encore as an experimental boundary. Based on feedback from users and our own intuitions, we will push the boundary as far down as possible to get the performance and functionality we need.

Jim Rumbaugh, GE R&D Center: I would like to hear a little more discussion about garbage collection on large databases. Of course, the easy approach is to let the user do it, but that has many disadvantages. I would like to know whether people think it can be made to work and how. Also how you do garbage collection, if the system is distributed

and must continually remain up, i.e., you cannot shut it down to make a garbage collection sweep.

Satish Thatte, Texas Instruments: How about not doing garbage collection at all as the solution to your (Jim Rumbaugh's) problem? You have only immutable objects, and you create new versions. Sure that would cost a lot of storage, but storage is getting cheap!

David Maier, Oregon Graduate Center: I am not going to solve your problem of distributed garbage collection. I think a real important thing is garbage prevention. That is, you are going to co-mingle temporary and persistent objects in a workspace. You have to do as much work as you can before committing to prevent the garbage from getting to the database in the first place. I really think persistence has to be on the basis of reachability from some persistent root, and then you garbage collection either incrementally or at the time when you commit them to the database.

Richard Steiger, Park Place Systems: I would like to ask David Wells how abstract data types help you with the dangling reference problem.

David Wells, Texas Instruments: That is pretty simple. You just ensure that the abstract data type does not give you a pointer into the middle of itself. We do not do garbage collection. Every committed object is in the database forever.

John Carnegie, Mentalix: I would like to ask David Maier if objects are going to change in size, how are you going to get around preventing garbage at commit time ?

David Maier, Oregon Graduate Center: I am not saying that all garbage is preventable. I am just saying that there ought to be smarts in the workspace manager that does not simply copy everything to disk at commit time. If you are not saving everything, you want to avoid storing anything to the database that will become garbage as soon as the transaction commits.

Eliot Moss, University of Massachusetts: I just wanted to see if I could convince Fred Brown (University of St. Andrews) to say something about the garbage collection in the persistent object support for PS-Algol, because they claimed to have implemented garbage collection on disk.

Fred Brown, University of St. Andrews: What we have basically done is to implement a two-level storage model because we cannot access disk efficiently. We have a local workspace and we do garbage collection there. We do not have to get very much garbage going out to disk.

Question from the audience: How can you increase query performance ?

Lawrence Rowe, University of California, Berkeley: To process a query presented as a string, first you have to parse it, validate it, then pass that all off to the planner. The plan gets handed off to the executor and the executor does operations of fetching tuples or pages from the disk and executing predicates on the tuples to determine what is going on. If you handle a query string at this level, it is guaranteed to go slow. So, what are the tactics for making it go fast? Well, one tactics is to cache the descriptor that describes how to go to the disk. The real way to go fast is to avoid all this junk, and precompute and cache the plan so that you can do a remote procedure call on the cached plan from an application.

The design alternatives are based on when the plans are created and where they are cached. One possibility is to compile the plans at the time the program is compiled, and put those plans on disk. This is what is done in DB2. So when a program is run, it says "run query 32," and at the time the query is submitted first, it goes out to disk, grabs the plan, loads it in memory, and runs the plan. I just want to point out that going to disk requires about 30 milliseconds, so the guaranteed shortest time to do this has got to be the overhead to get to disk, plus 30 milliseconds. The second alternative is to cache the plan in the DBMS. The first time you run the query you go through all this stuff, and then just leave the cached plan in the back end, so that for subsequent executions of the same query, all you have to do is pass the parameters, plug the parameters into the plan and execute it. That is done in RTI INGRES and it is a reasonably fast technique.

The tradeoff is the time it takes to go through this versus the time it takes to fetch from the database. The third possibility is to cache these plans in the application program. At the time the program is loaded, the plans come in at the same time. That is what is done in the new Tandem distributed SQL system. By putting plans in the application programs, you do not have to pay the time to do the first compile, and you also do not have to pay the time to go to disk the first time you want to run the query. What they effectively do is just glue in the cached plan into the application program and they do the remote procedure call directly to the back. The second issue is when do you validate the cached plans? Well, you have to validate the data types that are passed across. Then you also have to worry about schema changes. Again in System R, the plans were all stored in the database, if there was a schema change, they could just invalidate all the plans in the database. In the RTI world, validation for cached plans is still needed at run time, resulting into overhead.

Alfred Spector, Carnegie Mellon University: I must admit that in my world of on-line transaction processing, long transaction mentioned in design applications is completely foreign to me. We think in terms of maybe a quarter of a second. What I see is that transactions are relatively short, but there is some higher structuring that is needed in these design applications. There is user-level locking and user-level version maintenance associated with them; probably, it does not make sense to think of the low-level transaction system to maintain these very long term transactions. This may be is self-serving, because our system (Camelot) does not do it, but I really do not know what I would do if I had to do it. I would let people program the locks on top the system, and implement check-out/check-in policies on top.

How many people are implementing, planning to implement, or have implemented object-oriented database systems of some kind? If you are with someone else here, just one of you raise your hand. (At this point about 25 people raised their hands!) One thing I wanted to observe is that we are not reaching closure on this yet. This is still like the 60's when everybody was developing a new programming language and every Ph.D. thesis was another programming language notion. I do not believe that there is a common model that is going to emerge for next ten years in this area. So there is not going to be a standard. We are going to have to have systems that are inter operable. We are going to need to access RTI's INGRES at the same time to access a lot of data in it, and we need to access

our special purpose systems as well. They are going to have to work together, because there is no alternative. We are not going to have a common single system that everyone is going to use. The only thing that I know is that the operating systems people should really try to do something to help; otherwise there is going to be 10 times 20 man years of work built on the low-level components, as everyone re-extends UNIX to do exactly the same low-level primitive functions.

4 Conclusion

There were clearly two camps at the workshop. The majority camp was coming from the programming language community. It wanted to extend programming environments with persistent object repositories and object-oriented databases (not surprising, as this was the OOPSLA '87 conference). The minority camp coming from conventional database community, although shared many of the goals of the majority camp, differed markedly in its approach (extending a relational database) and emphasis (query optimization versus unpredictable navigation in the object world). However, many participants felt that new equations are emerging in the market place where there will be room for conventional databases to handle massive business data and complex ad hoc queries, object-oriented databases to handle and navigate engineering data with complex structures, and databases for real-time transactions to deliver high throughput for simple transactions. A new market segment is likely to emerge that will help tie these heterogeneous databases.

It was evident from the workshop and the conference that the field of object-oriented databases is taking off almost exponentially as a strong market-driven activity. Over 25 efforts are currently underway to implement OODB systems. There are many difficult research problems that need to be solved: object-oriented data models, management of composite objects, OODB programming languages, distributed transaction management on abstract data types for co-operative design environment, change management for evolving objects, object sharing in multi-lingual and heterogeneous distributed environments, query optimization techniques for abstract data types, development of an appropriate performance matrix for OODBs, and performance and reliability issues. Many workshop participants expected that major progress will be made to solve these problems over the next five years. Relational database technology took over ten years before being accepted by the market. OODB technology will need five to ten years to transition from its current status of "proof of concepts" to the status of "full commercial systems."

Based on the written evaluation forms returned by the participants, the workshop was a success within the constraints of its three hours time limit. However, most participants would have liked a better focused, full one-day workshop, with an advanced list of issues and agenda distributed to the panel members. This is an excellent suggestion, and should be considered for the OOPSLA '88 conference to be held at San Diego, CA.