PCLOS: A Critical Review

Andreas Paepcke
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
paepcke@hplabs.hp.com

Abstract

This paper uses the persistent object system PCLOS to survey some problems and benefits of object persistence. The system is analyzed along several relevant dimensions. PCLOS provides object persistence for an object-oriented language. The insights gained on desirable and detrimental components of the system are presented. The intent is to outline some of the expected and unexpected problems encountered in the construction of support for object persistence.

1 Introduction

PCLOS (Persistent CommonLisp Object System) provides object persistence for an object-oriented language [1]. It has been stable for some months now and several test applications have used it. While usage is not widespread enough to support substantial statistical observations, a large amount of insight has been gained on desirable and detrimental components. Some of these have been documented by various researchers [2, 3], while others have not been reported formally.

While reference [1] describes PCLOS itself, this review documents some triumphs and failures of the system. The intent is to draw attention to some obvious and to other more subtle problems in the context of object persistence. We do include difficulties that are specific to the current implementation. While these might not be fundamental to the concept of object persistence, they are often symptoms of problems whose solutions are conceptually simple but that are difficult to implement. Some other non-fundamental shortcomings can point out traps that can be avoided in other implementations. We hope to provide information that can be of use in the construction of any object persistence machinery. PCLOS is used as a road map through the relevant issues.

The following section will provide a very quick overview of PCLOS. Each of the remaining sections then analyze one component of the system. In the first part of each section we generally explain the motivation and advantages of the component. This is then contrasted with descriptions of problems we encountered. Such problems include weaknesses in the implementation, fundamental technical problems and conceptual difficulties experienced by users.

2 Outline of PCLOS

PCLOS stands for Persistent Common Lisp Object System. It provides persistence for objects in the Common Lisp Object System (CLOS) [4]. PCLOS is organized to allow the simultaneous use of multiple, different database management systems. Figure 1 shows the architecture. The language and program layer is insulated from differences among the databases by a *virtual database*. This implies the necessity of two mappings: objects and classes in the language layer must be described in terms of operations and concepts of the virtual database. The second mapping expresses the virtual database concepts and operations in terms of the various specific database managers.

The data model of the virtual database is 'relationesque' in the sense that it is based on *virtual tables*. The model is not fully relational because fields may reference rows in other tables. Each programming language class is *conceptually* represented by a table whose columns correspond to the slots[1] of the class and whose rows correspond to the class' instances. An object can therefore be uniquely specified by providing information about the database it resides in, the name of the appropriate virtual table and the virtual row number.

The mapping of the virtual tables onto database management systems obviously depends on the concepts and interface details of those systems.

The virtual database's model is just complex enough to accommodate the modelling needs of object-oriented languages. It is no more complex than this because it must be possible to implement it on very simple data stores. This design principle of minimal complexity is compromised by the desire to take advantage of very *intelligent* database management systems. An inflexible minimal virtual database model would preclude access of the language layer to such advanced capabilities.

PCLOS attempts to solve this problem by the notion of *protocol adapters*. Each of these adapters represents one database concept that is not represented by the virtual database's *core protocol*. Examples are transaction management, database archiving or complex queries which provide existential quantifyers and multi-valued returns. Attempts to use these features on database systems that do not support the corresponding concept will cause a *protocol mismatch* error.

Objects move from databases into the language layer either because they are in the solution set of a query, or because a reference to them caused an *object fault*. When an object is *internalized* in this way, PCLOS creates a *husk object*, an in-memory data structure representing the object for purposes of message dispatching and in-memory referencing. Slot accesses to husk objects are trapped and translated into updates or queries

[1] The CLOS *slots* are called *instance variables* in many other object systems.

Figure 1: Design Model

to the appropriate database. When an object is *cached*, its slot values are brought in from the database and are placed in the husk. From then on access to the object's slots are handled like accesses to non-persistent objects.

The remaining sections examine some details of this system. Additional information on PCLOS mechanisms will be introduced as needed.

3 The Architecture

In this section we look in more detail at the PCLOS architecture that was outlined above. It is, of course, motivated by the goal of supporting the use of different database management systems and it has succeeded in this goal.

Beyond that, it has strongly encouraged modular and object-oriented programming. Low-level database interfaces, for example, are implemented as classes whose instances understand database-specific details. Interfaces between the program layer and virtual database sessions are represented as instances of another class.

Another indirect benefit of the architecture is the fact that its implementation produces as 'fallout' a concise, complete checklist of concepts and features that must minimally be provided by all database management systems to be accessed through PCLOS.

Ironically the same aspects that make the architecture successful overall, are the source of drawbacks: the strict insulation between layers forces duplication of administrative data and therefore leads to wasteful use of memory. To illustrate this, let us look at the access to the data that comprises the state of some object. At the language level this access uses as an access specifier a machine pointer to the object husk. At the virtual database level, the corresponding object is accessed by using a table name and row number as an identifier. The physical database, finally, will generally use its own ideosyncratic identifier. This is typically an object identifier generated by the database, or a tuple identifier. In order to ensure the desired modularity, the system's implementation layers should not know about object identification details of the other layers. The language layer must therefore maintain a mapping between machine pointers and table-row pairs. The virtual database layer in turn must maintain a mapping between table-row pairs and some database access key. In fact, this layer does not understand the concept of an object at all. It deals exclusively with virtual tables and rows. All of these mappings must be reciprocal to facilitate movement of objects both into and out of the databases. A less flexible, more monolithic architecture would require fewer mappings.

4 Object Modes

PCLOS, like [5, 6, 7], provides a number of ways for an application programmer to control the location and sharability of objects at run-time. These options position the object on a three-way tradeoff of access speed, amount of concurrency and crash resilience. Here are outlines and brief discussions of these options.

Object-level Persistence

Objects can be persistent or transient. The database is not aware of the existence of transient objects and they cannot be shared. Transient objects may be made persistent and vice versa. Both transient and persistent object behave like ordinary objects in the context of message sending.

Slot-level Persistence

Objects may be persistent as a whole, but individual slots may be declared to be transient. This option may be used when some slot values are not meaningful across sessions, or when clients sharing the objects should each have private copies of these slots. All other persistent slots are subject to concurrency control. This is supported both for instance-allocated and class-allocated CLOS slots. The latter are slots which are shared among all instances of a class.

Uncached Operation

All accesses to the persistent slots of a persistent object will by default involve database operations. A read will fetch a value by submitting a query which may be precompiled for improved efficiency. A write will cause a database update operation. Unless a programmer has specified that a transaction be started, each of these accesses will be a small, complete, committed transactions. This mode of operation maximizes concurrency of object accesses, but it has the highest performance cost. Note that for in-memory or heavily buffered object stores this cost is not necessarily prohibitive.

Object-level Caching

Objects may be cached by sending them an appropriate message. This may be done within a transaction, in which case access to the object by other processes is blocked. If the cache instruction is issued outside of a transaction, the in-memory copy is a snapshot.

Slot-level Caching

Individual slots of persistent objects may be cached. This allows speedy access to those slots while maximizing concurrency of access to the others.

Write-through Cache

Finally, a write-through mode causes the database to be updated whenever an object slot is modified. Slot reading, on the other hand, will return the memory-resident value. This causes updates to be propagated quickly, while providing good performance for reading. The transaction mechanism may be used to guarantee consistency of in-memory data. Applications like browsers may prefer to refresh the in-memory data explicitly upon request by the user.

Since PCLOS is an experimental system, caching has been left to the initiative of the programmer. The intent was to learn about common caching patterns through the use of these primitives and to provide caching automation based on the results. It became obvious very quickly that access to a full-functionality, disc-based database was too expensive for non-cached operation. Users involved with these kinds of databases frequently ended up caching indiscriminately. They soon requested more sophisticated ways of specifying objects to be cached and suggested some way to define 'cache groups' that would specify sets of objects that should always be cached together. The system described in [5] intends to provide such a mechanism in the form of a *hints* relation. Caching is less of an issue for databases that use caching techniques themselves or are memory-based with occasional dumps to files to provide persistence.

The availability of these object modes provides flexibility needed to tune operations, particularly for the needs of applications that involve sharing. Unfortunately, such flexibility can be confusing. Ideally, the system would be able to find the optimal mode to use. This would minimally require the ability of the underlying database to notify interested clients of changes. Much of the information needed to make an optimal choice among the modes is unfortunately based on application domain knowledge. Examples of such knowledge are "it does not matter whether the slots of this object are always up to date" or "this object will be read frequently, while updates are rare".

A compromise between the manual selection of modes and fully automatic operation could be to allow the programmer to *declare* the object properties needed as the basis for object mode selection. We plan to experiment with this notion.

5 Language Integration

The term *language integration* is used here to describe the degree of naturalness and elegance with which database concepts are introduced into a programming language and to which extent persistence-related details are transparent to programmers.

The advantages of full language integration are rather obvious. Adding new capabilities to a computing environment leads to a cost-benefit analysis. The need to learn new concepts is a cost and should therefore be minimized. A slightly more subtle point becomes apparent if we think about the alternatives to integration. These have generally been the introduction of embedded languages [8, 9, 10]. With this technique all aspects of persistence are controlled through library calls and no attempt is made to assimilate persistence-related concepts into the language paradigm. One drawback of this solution is that the programmer must assume much more responsibility in a realm that is new to him. The larger number of control options alone tends to make systems with embedded languages more error prone than a more integrated system.

PCLOS language integration was accomplished through the use of the CLOS metaobject protocol [11]. This mechanism is ideally suited to experiments in language integration.

It is based on the idea that after some bootstrapping, the object model of an object-oriented language can be implemented using that language itself. Employing the principles of object-oriented programming, the object model or its implementation may then be modified. The addition of persistence, for example, requires that all slot accesses be intercepted. PCLOS accomplishes this through the use of the metaobject protocol: the capabilities for representing and manipulating slots is inherited from the standard CLOS implementation. The slot access methods alone are then shadowed by methods that add persistence-related behavior.

Nevertheless, PCLOS suffers from a high number of transparency breaches. Some are a consequence of the specifics of the CLOS language, others are inherent problems of persistence. We outline some of these issues in the following subsections.

5.1 Type Mappings

When a PCLOS slot value enters into some database, the value's type and representation must be converted to some equivalent database-native type and representation. PCLOS accomplishes this through so-called *converter objects*. They have the ability to map almost all of the common language types. Programmers may extend this capability by providing conversion functions which are called when values are to be entered into a database. These functions must return an item whose type conversion is known. The programmer can also define the inverse functions that convert database values to an appropriate in-memory type and representation. POSTGRESS [6] provides a similar mechanism. Examples of types that are usually easy to convert to database-native types are strings and integers. Arrays are more difficult to convert for most databases. Some datatypes are very closely tied to specific run-time environments and do not trivially make sense when introduced into another environment. Lisp function objects are an example of this. In other languages problems would similarly be encountered with non-relocatable code segments. If some slots will contain such items, the programmer must either provide a converter function that turns the entry into some harmless dummy, or he must declare the respective slot to be transient. Both [6] and [12] allow programmers to register procedures with a database. These procedures are run by the database server as part of retrieving some specified value. Section 8 points out some possible shortcomings of this approach.

Type mappings are therefore one area where persistence transparency can break down.

5.2 Instance Initialization

When a new instance is created in most object-oriented languages, some well-defined sequence of initialization operations will be performed. Programmers understand that such initialization is done exactly once, at a predictable time and in a predictable run-time context. Object persistence changes these basic facts. Instances enter an address space not only because a programmer has placed a `make-instance` at some well-defined point in a program, but also because the instance was found through a query or an object reference at some random time within a non-predictable run-time environment. This situation is very different from traditional operating environments and it is not reasonable to simply apply the language's instance initialization protocol. There are several reasons for that:

When creating a CLOS object through the use of `make-instance`, the caller may add specifications of initial slot values for the new object. When an object is instead brought in from the database, these specifications are not available.

Another difficulty will arise when actions must be performed exactly once at the creation time of an object. Consider, for example, a computer configuration management program. Configurations are represented as objects which must be registered with the bookkeeping parts of the system. This registration should happen at the time a new configuration object is created. But it should not happen everytime some query retrieves the configuration object from the database.

The existence of transient slots is another reason for needing a different initialization protocol for objects being brought in from the database: while the creation of an object might call for the initialization of all slots, the persistent slots should generally not be modified in objects about to enter from the database. The transient slots, on the other hand will need to be initialized.

For these reasons PCLOS guarantees that the method `after-retrieve` is called exactly once on every object immediately after it first enters a program's address space from the database. This method may be used by programmers for the purpose of initialization.

`After-retrieve` methods do solve the persistent instance initialization problem, but their existence obviously violates the transparency goal. It was indeed a source of user confusion in surprisingly many cases. The problem was usually that an appreciation of instance initialization in this context requires an understanding of how and when objects move between databases and run-time environments. The transparency in other parts of the system tends to induce a false sense of safety in dealings with persistent objects.

5.3 Data Location Stability

We define the term *data location stability* to mean that run-time data structures are either guaranteed to remain in one physical location in memory throughout one session, or that all references to the structure are guaranteed to be updated if the data is moved by the system. CLOS – like other languages – distinguishes between two pointers referencing distinct data items which might have identical contents, and situations in which two pointers point to the same physical place in memory. Programmers often rely on this notion. There are two operational contexts in which PCLOS does not preserve such location stability for slot values that are neither object references nor symbols.

5.3.1 Cache Rollback

When objects are cached into a client's address space, PCLOS provides transaction rollback which would normally be available only for data that is under database control at all times. To do this, PCLOS makes a deep copy of a slot value whenever a program tries to update the slot for the first time since a transaction was started.

After the copying has been completed, programs continue to operate on the cached objects. In particular, a program might for example overwrite a slot's reference to some data structure. With the reference gone, automatic or programmer initiated garbage collection could destroy all traces of that structure. A subsequent state rollback would cause that slot to point to a very similar-looking data structure, but it will not be at the same location as before the transaction began. Note that the same problem exists in other languages as well. A C program could allocate space for a data structure and subsequently begin a transaction. The program could free the storage within the transaction. A rollback could recreate a data structure with identical contents, but the space allocated for it will generally not be at the same physical address.

This can lead to program errors, if the destroyed structure was referenced by some other object and location stability is assumed. This problem could be solved only at the cost of doing most of the work done by a garbage collector because the system would have to find every reference to every data structure it saves. An alternative might be to postpone all storage reclamation until transaction commit. That solution could be quite complex.

5.3.2 Database-Resident Information

When PCLOS accesses an object's slots without caching the object, structure sharing among slots is not possible. This implies that a programmer cannot rely on consistency of multiple references to a data structure[2]. The reason is that when a slot is accessed, the database representation of its value is retrieved from the database and is converted to a correctly 'shaped' in-memory data structure. Storage is allocated for the structure as part of the retrieval. Again, the acquisition of knowledge necessary to preserve location stability would require a substantially modified storage management. All data structure sharing would have to be maintained in the database. Reference [13] shows how this issue could be tackled.

In summary we see that data location stability would not be impossible to achieve. But in this kind of environment it can be accomplished only at significant cost of run-time performance or of restrictions placed on programs.

[2]In Lisp this is the concept of *eq*-ness.

5.4 In-Place Data Manipulation

The problem described in this section occurs when a programmer modifies some slot's data structure in-place without caching the respective object. As outlined in the PCLOS introduction above, a slot value of an uncached object is copied from the database into the runtime environment transparently when the slot is accessed. This is done by intercepting the slot access mechanism. The database is updated transparently when a slot value of an uncached object is explicitly changed by some assignment operation. Again, this is done by intercepting the language's slot update mechanism. This is quite satisfactory in many cases. But it fails if a slot value is changed without an explicit assignment. An example would be the in-place modification of an array element. This does not involve an explicit slot update and will therefore not cause a database update. PCLOS users who want to use this mode of operation have introduced local variables at the beginning of their methods. These are initialized to hold slot values that will be modified in-place in the body of the method. At the end of these methods, the respective slots are updated using the regular slot assignment mechanisms. The problem can therefore be avoided by appropriate programming habits, but that certainly compromises transparency. To clarify, the problem does not come up when objects are cached, modified in any way and unconditionally written back to the database. Some object servers operate using this procedure exclusively [14].

The problem is unfortunate, because the mode of operating without caching allows for the maximum amount of concurrency in the sharing of objects. The issue is fundamental, because many languages allow users to define their *own* in-place operations. It is therefore not possible to guarantee that all in-place operations cause a write-back to the database.

6 Virtual Database Specification

The specifications for the virtual database can be surprisingly simple. The table model described earlier is straight forward and sufficient in many ways. Two main issues are of interest when evaluating this model.

6.1 Storage Granularity

The virtual database model commits the granularity of the data stores' structual 'awareness' to the level of objects.

By this we mean that only objects have a unique identity in the database. There is no identification of the objects' slot values. The database does not understand the structures of slot values stored in it. Arrays, for instance, are encoded to look like strings in the database. Another example are Lisp *symbols* which are also not modelled at the database level.

This is most noticable in the context of querying. It is, for instance, not possible in PCLOS to ask for objects whose slot **foo** contains an array whose third element is equal to ten. Object-oriented databases are now trying to model programing language structures [12]. This could solve such problems if an efficient implementation is possible.

6.2 Inheritance

The second issue to consider in the evaluation of the PCLOS virtual database model is the absence of an inheritance notion. Recall that a language class is modelled as a virtual table whose columns represent slots. When PCLOS creates such a table, it provides columns for the union of slots defined by the class itself and by its ancestors. Alternative approaches would make the virtual database's model fully object-oriented, or would introduce inheritance by allowing the declaration of parent-child relationships among virtual tables. The data comprising the state of an object would then be spread over columns in several virtual tables. The advantage of the PCLOS approach is its simplicity and improved performance in many cases. Simplicity is important because it allows implementations on very simple object stores. Performance is often improved because many table-oriented databases allow explicit instructions for *clustering* data in tables. Distributing data over several physical tables would mean that PCLOS could not take advantage of such clustering in the implementation of the virtual database.

Another performance benefit is the special ability of some databases to retrieve and update entire tuples in physical tables. In implementations of the virtual database that use these physical tables such streamlined special access paths benefit update and retrieval of entire objects. These two operations are obviously frequent.

The simplicity and performance advantages of the PCLOS virtual database model do have costs. The model implies that columns are created for slots at each level of the class hierarchy. If class **A** contains slot **foo** and class **B** inherits from **A**, the virtual table representing **A** will contain a column for **foo**. The virtual table that represents **B** will also contain a column for **foo**. If the implementation of columns in virtual tables is expensive, the model could be suboptimal.

Another disadvantage of the PCLOS virtual database model is encountered when class redefinition is to be supported. Both [12] and [15] in the database community, and [4] in the programming language community point out that such evolution does need to be supported. When, for example, a slot is added to some class, the associated virtual table must be changed to contain an additional column. If the virtual database featured inheritance, one such table modification would be sufficient. Without inheritance notions in the model and its implementations all the virtual tables representing children must also be modified.

In summary, the simple model of the virtual database has benefits and costs. Experience with applications of PCLOS suggest that the benefits outweigh the costs.

7 Database Requirements

The design goal of providing unified access to multiple object stores with varying capabilities implies that the requirements on the database management systems should be as non-demanding as possible. The simple virtual database model has generally been successful in this respect. We did, however, encounter several unanticipated, annoying restrictions when working with real database systems. The following subsections explain these problems. Any specific database management system may exhibit some or none of these shortcomings. This discussion is meant to draw attention to possible problems in the implementation of the virtual database on various available databases.

7.1 Typing

Inelegant ad hoc techniques must sometimes be used if an essentially untyped language like CLOS is made persistent on a strongly typed database. PCLOS will use any type information a programmer provides in the context of class definition. But since such type information is optional in CLOS, the system does often need to revert to such techniques. The implementation of virtual fields must allow values of different types in those fields. There are two solutions an implementation might try to avoid type mismatch errors.

The first is to find a way to represent all data in terms of one type. PCLOS does this by encoding data as strings when necessary. The use of this technique is sometimes unavoidable, but there are obviously performance penalties to be contended with.

Another drawback of this technique is that the potential advantages of a perfect type match between language and database are lost. Values of type `integer`, for instance, may often be stored as database-native items. When databases are 'aware' of the special characteristics of items with a particular type, storage performance and functionality gains may be provided by the database. One example of this could be the availability of relational operators like `greater-than` or `less-than` in queries over fields known to be restricted to `integers`.

A further potential advantage of having databases be conscious of value types is that conversions of items in the database to data structures in multiple languages and vice versa might be provided by the database implementators. Sharing of objects across language boundaries is therefore facilitated by the use of database-native types.

A technique other than the encoding of diverse data items to values of a single database-native type may be employed if the model of the database used provides *type hierarchies*. In this case the implementation of virtual fields may specify that all future values stored will be of a database-native type which is high enough in the type hierarchy to include the types of these values as children types.

An object-oriented database may, for instance, have a type hierarchy whose root is the very generic type *object*. Below this root type there might be a child type `literal` which in turn has children `string`, `integer` and others. If the programmer has not specified a type for some slot, the database storage type for the slot can be declared to be `object` which is general enough to be equivalent to untyped operation. But if the programmer has restricted a slot's value to be, for example, an `integer`, PCLOS will pass this knowledge down to the database when appropriate.

7.2 Physical Data Representations

While most data representation differences between databases and programming languages can be taken care of, this issue can sometimes become an unexpected road block. For an example, recall that PCLOS encodes as strings all data whose type does not have an equivalent database-native or constructible type. As might be expected, this encoding contains the lengths of encoded data items, values of integers in arrays and other information. This works well – unless a database uses null-termination in its string type implementation! While this conflict can be resolved by appropriate data encoding techniques, it can be a time consuming and unexpected annoyance.

7.3 Concurrency Control

The PCLOS virtual database does not model concurrency control. It rather relies on the mechanisms of the underlying databases. One reason for this is that we felt the need to build a system and to learn about concurrency control issues by practical experience. A more technical reason is the fact that most databases perform concurrency control at a very low level and deeply inside the system implementation. Providing a uniform model of concurrency control would therefore require either unacceptable modifications of the various databases to be used, or it would suggest that the built-in concurrency control mechanisms would not be used at all and that this job would be accomplished by software above the database level.

7.4 Other Typical Problems

Our use of PCLOS has confirmed the existence of classical problems reported by others [2, 3], and has exposed additional, less publicized difficulties. The system does not solve the following problems:

- Long lasting use of protected resources.
- Large lock granularity.
- Problematic error resolution policies.
- Insufficient resource access control mechanisms.

The long lasting use of shared resources has been described in several places, including [3] above. The others will be outlined in the following sections.

7.4.1 Error Handling

An area in which this new database application domain of object persistence requires revisions of traditional database designs is the systems' handling of error conditions. The action taken by some databases when a deadlock is detected may serve as an illustration of this point. When interactions with databases are short, self-contained sequences of operations which are to be viewed as atomic, it makes some sense for a database management system to simply abort the operations of one of the clients involved in a deadlock. The aborted party does not need to manage substantial context information to resubmit the sequence of operations. If, however, a sequence of interactions between database and applications involves hundreds of operations, like a document editing session, a random abort by the database management system is unacceptable. In such a case the applications involved should instead be notified of the deadlock condition. They would then presumably negotiate and save as much as possible of the work done.

Another solution might be to enable a database to resubmit a transaction.

7.4.2 Concurrency Control Primitives

Insufficiently rich concurrency control primitives are another issue in the context of object persistence. In many cases applications in this domain would like to be notified when some specified changes occur in the database. Notification locks or general trigger mechanisms are useful for such cases. Another example is the fact that applications like object browsers might want to present objects, even though another application is modifying them. Such *dirty reads* might appear dubious for traditional applications, but they are very reasonable in other contexts.

We see that the area of concurrency control poses difficult new problems. This is particularly unpleasant for implementors of object persistence because, as mentioned before, concurrency control tends to be implemented far down in the database system implementation. This is most likely where it belongs, but it makes experimentation difficult.

8 Query Capabilities

Query capabilities and their introduction to programming will potentially be very powerful. At the same time they introduce several rather fundamental and sometimes uncomfortable questions.

The ability to find data structures by associative search through their contents is new for programming languages in general, and object-oriented programming in particular. At first glance, objects seem to be prime candidates for such a capability because they are small, manageable chunks of regular structures that together embody all of the application data. It is, however, clear that such a capability cannot be reconciled with the object-orientation concept of *encapsulation* if the query accesses the state of objects directly.

A resolution of this conflict can be approached by ensuring that queries are formulated in terms of method invocations alone, as opposed to direct access to data which is encapsulated by objects. This approach would be much closer to satisfying the notion of encapsulation. It is, however, a potentially interesting question whether the mere capability of finding all instances of classes violates some very strict understanding of encapsulation.

While this encapsulation issue could potentially require a choice among these extremes, solutions to another query-related issue may be more flexible. This issue concerns the amount of structural knowledge a query processor is expected to have. The understanding of inheritance relations among classes and consequently their storage representations is a prime example. An informed query processor might qualify instances of child classes as query solutions, whenever instances of parent classes qualify. A less informed processor would not understand such subtleties.

9 Performance

When analyzing perceived performance of object persistence, several factors must be taken into account. Numbers of "objects retrieved per second" are close to irrelevant to evaluating everyday use of the systems or to compare them, unless many of these factors have been taken into account:

- How large are the objects?
- Are object manipulation and object storage management done in separate processes?

- Are network transfers included in the object flow?

- Are object transfer rates based on some tightly controlled, highly optimized, steady-state experiment, or were the measurements taken under realistic conditions where object flow and object-related computation are interleaved? Such realistic conditions can cause additional process swapping and query re-optimizations which would not be taken into account in a more 'purified' environment.

- Did the measurements include object slot values that required data manipulations to accommodate formats not modelled natively by the database?

On top of a full-functionality disc-based database PCLOS exhibited speeds of about one object per second for objects with an average of twenty slots whose values translated to an average of 300 bytes. This measurement was made in an environment where database management was done by a separate process on another machine through an ethernet local area network. The measurements were made in the context of applications which generated slot values that did require data encoding in most cases. The system was much faster with object stores that kept major or all portions of objects in memory, even when management was done on another machine. These configurations are obviously much less resistent to crashes, but they do allow data sharing.

10 Related Work

More and more researchers are becoming interested in various issues of object persistence. This section offers a brief survey under various criteria and provides references to relevant literature.

10.1 Language Support

Some systems, such as ORION [15, 16, 17, 18, 19], GARDEN [14, 20] and Coral3 [21], propose the use of one language to program all applications. Single-language systems differ from each other in that some propose entirely new languages (e.g. FAD [22]), while others attempt to extend established ones. The advantage of single-language systems is that persistence may be integrated more easily into one language than into a layer of persistence that must be added to an arbitrary number of languages. The advantage of multi-lingual systems is obvious.

10.2 Persistence Integration

When evaluating the degree to which persistence is integrated into a programming language, it is important to note that complete transparency is not possible, because the addition of persistence introduces new concepts which will have to be reflected in the programming language. Examples of such concepts are transactions and queries. Systems do differ widely in their approach to integrating these extensions into the languages they support. PS-algol [23, 13, 24, 25, 26, 27, 28] provides system functions, and PCLOS similarly uses methods and inheritance to add persistence related capabilities to a language. Systems like GemStone [29, 30, 31] rely on remote procedure calls to a virtual data processor, while *ts1* [32] and others have modified the syntax of a language such as C by means of preprocessors.

An important difference between levels of persistence integration surfaces in a system's approaches to controlling persistence itself and to managing object caching. While PS-algol, VBASE [33, 34] and GARDEN automatically make all objects persistent upon creation, PCLOS and FAD [22] are examples of systems that distinguish between persistent and transient data. GARDEN provides fully automatic caching, while the Shared Object Hierarchy [5, 6, 7] and PCLOS provide many different options to control data traffic between main memory and persistent storage.

In all of these aspects of persistence integration, the advantages of transparency and syntactic elegance must be weighed against needs for performance and portability.

10.3 Concurrency Control

Systems vary in their facilities to control concurrent access to objects by multiple users. The three main techniques are time stamping (e.g. *ts1*), locking (e.g. Coral3) and transactions (e.g. ORION). Time stamping systems will refuse to update a data item if the current database copy of the item is newer than the copy that is intended to replace it. This scheme has the advantage of being simple. One disadvantage is that complex operations requiring many updates might fail part way through, leaving the database in an inconsistent state. Even if only one data item is involved in an operation, race conditions cannot be avoided.

Locks will solve the problem of racing, but will still not provide atomicity of complex operations. Systems using locks for concurrency control differ in the sophistication of available locks. *Read* and *Write* locks which control read and write access are common to all systems. *Notification* locks, which generally allow object updates but will notify interested parties, are found less frequently (e.g. Coral3, GARDEN).

Transactions are the most sophisticated method of concurrency control. In addition to providing atomicity, they acquire and release data locks automatically in a safe manner, and provide rollback, the capability to abort a sequence of operations. This will "roll the database back" to the state it was in before the transaction began. Some transaction mechanisms allow for a "redo" which repeats previously executed operations. Another advantage of transaction facilities is that they often provide deadlock detection which must be handled by the programmer in systems that only offer locks.

11 Conclusion

PCLOS has contributed significantly to our understanding of issues pertaining to object persistence. Its strong emphasis on provision of unified access to multiple, heterogenous databases has proven to be extremely valuable. It enables application designers to switch among various databases to get support for object persistence. The generous caching options add additional flexibility, although some of them would probably be removed in a production system which would manage at least some of the caching process automatically.

The disadvantage of this general, flexible approach in contrast to a tight integration of object storage and object computation is the fact that services like querying, concurrency control or administrative support cannot be optimized. But because of the varying needs of different applications we believe that the cost of allowing multiple database management systems as underlying storage is outweighed by the benefits of this approach.

While PCLOS succeeds in providing good integration of persistence when compared to embedded database control languages, several aspects of persistence could not be hidden. Major examples are instance initialization for objects obtained through queries, break-downs of storage location stability and object-based waste management.

Performance is a problem that must be addressed. We feel that the most likely ways to overcome the problem are memory-resident or heavily buffered object stores which provide sharing and consistency support and use a simple database as a back end.

Object persistence is a fast growing technique, and its potential benefits are substantial. The topic is rich in diverse problems ranging from nuts and bolts implementation tricks to questions of compatibility among concepts that evolved over years of research in the separate areas of programming language design and principles of database management systems.

12 Acknowledgments

Brian Beach, Joe Mohan and Jim Kempf have laid the groundwork for PCLOS with their design and construction of DOOM [35]. Brian Beach has gone on to build an in-memory, multi-user, archivable object store which can be accessed through PCLOS. The performance of this object store has provided hope for the feasibility of object persistence as a concept.

Discussions with the members of the Iris project at Hewlett-Packard Laboratories have helped me understand the database community view of the world. Beyond that they have provided me with a particularly interesting database to use with PCLOS.

Mike Creech and Dennis Freeze have patiently provided me with one mean, circular, recursively defined test case after another. Their help in the analysis of error conditions has been crucial in making PCLOS usable.

A 'second generation' of friendly users consisting of Lucy Berlin, Bob Leichner and Vicki O'Day has begun to provide larger scale stress tests for object base size, performance and functionality.

All of these people have sacrificed progress in their own projects to provide the help I needed.

References

[1] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer Verlag, 1988.

[2] D.C. Tsichritzis and O.M. Nierstrasz. Fitting round objects into square databases. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer Verlag, 1988.

[3] Philip A. Bernstein. Database system support for software engineering. In *9th International Conference on Software Engineering*, 1987.

[4] Daniel G. Bobrow, Linda DeMichiel, Richard P. Gabriel, Gregor Kiczales, David Moon, and Sonja Keene. *The Common Lisp Object System Specification: Chapters 1 and 2*. Technical Report 88-002R, X3J13 standards committee document, 1988.

[5] Lawrence A. Rowe. A shared object hierarchy. In Klaus Dittrich and Umeshwar Dayal, editors, *Proceedings of the International Workshop on Object-Oriented Database Systems*, Association of Computing Machinery, 1986.

[6] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES data model. In Peter M. Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases*, Morgan Kaufmann Publishers Inc., 1987.

[7] Michael R. Stonebraker. The design of the POSTGRES storage system. In Peter M. Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases*, Morgan Kaufmann Publishers Inc., 1987.

[8] C.J. Date. *An Introduction to Database Systems*. Addison Wesley, Reading, Ma., 1983.

[9] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[10] M.M. Astrahan et al. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[11] Daniel G. Bobrow and Gregor Kiczales. *The Common Lisp Object System Specification: Metaobject Protocol.* Technical Report 88-003, X3J13 standards committee document, 1988.

[12] D. Fishman et al. Iris: an object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, April 1987.

[13] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[14] Andrea H. Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object-oriented database system. In Klaus Dittrich and Umeshwar Dayal, editors, *Proceedings of the International Workshop on Object-Oriented Database Systems*, Association of Computing Machinery, 1986.

[15] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of the ACM Special Interest Group on Management of Data*, Association of Computing Machinery, 1987.

[16] Jorge F. Garza and Won Kim. *Transaction Management in an Object-Oriented Database System.* Technical Report ACA-ST-292-87, MCC, September 1987.

[17] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Composite Object Support in an Object-Oriented Database System. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1987.

[18] Won Kim, Jay Banerjee, Hong-Tai Chou, and Jorge F. Garza. *Object-Oriented Database Support for CAD.* Technical Report ACA-ST-293-87, MCC, September 1987.

[19] Darrell Woelk and Won Kim. Multimedia information management in an object-oriented database system. In Peter M. Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases*, Morgan Kaufmann Publishers Inc., 1987.

[20] Stephen P. Reiss. *GARDEN: An Environment for Graphical Programming.* Brown University, October 1985. Reference and Programmers Manual.

[21] Thomas Merrow and Jane Laursen. A pragmatic system for shared persistent objects. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1987.

[22] François Bancilhon, Véronique Benzaken, Claude Delobel, and Fernando Velez. *The O_2, V0 Object Manager Interface.* Technical Report Altaïr 11-87, Altaïr, BP 105; 78153 Le Chesnay Cedex; France, September 1987.

[23] Malcom Atkinson et al. PS-Algol: an Algol with a persistent heap. *Sigplan Notices*, 24–30, July 1982.

[24] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1983.

[25] M.P. Atkinson and R. Morrison. Integrated persistent programming systems. In B.D. Shriver, editor, *Proceedings of the 19th Annual Hawaii Conference on System Sciences*, pages 842–854, , , 1986. Vol. IIA, Software.

[26] A.J. Cole and R. Morrison. *An Introduction to Programming with S-algol.* Cambridge University Press, 1982.

[27] R.L. Cooper, M.P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing database systems in a persistent environment. In Peter M. Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases*, Morgan Kaufmann Publishers Inc., 1987.

[28] *PS-Algol Reference Manual - Third Edition.* Universities of Glasgow and St. Andrews, 1987. Persistent Programming Report 12.

[29] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data*, 1984.

[30] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1986.

[31] D. Jason Penney and Jacob Stein. Class Modification in the GemStone object-oriented DBMS. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1987.

[32] Michael Caplinger. An information system based on distributed objects. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1987.

[33] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Association of Computing Machinery, 1987.

[34] Ontologic, Inc. *Vbase Integrated System Technical Overview.* Technical Report, Ontologic, Inc., March 1987. Technical overview for revision 0.8.

[35] Brian Beach and James Kempf. *DOOM: Permanent Objects for Common Lisp.* Technical Report STL-TM-86-09, HP Labs, September 1986.