Persistent Programming Languages: The Best of Both Worlds

Rex Jakobovits
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195

December 9, 1993

Abstract

The integration of databases and programming langauges is being motivated from two directions. The database community requires a more flexible and powerful way of modeling the world, whereas the programming language community wants the convenience of a reliable, efficient means of enabling entities to persist between program invocations. Traditionally, the query facilities provided to database users are not computationally complete, precluding arbitrarily complex processing of data. Furthermore, they support only primitive data types, making them inappropriate for modeling certain real world applications. Processing must be done off-line in a host language, but translation between the database and the language results in an impedence mismatch problem. One solution is to extend an existing programming language with the notion of persistence, enabling it to seamlessly interact with the storage manager. This paper is a survey of such efforts and the issues involved, focusing primarily on persistent object-oriented languages.

Contents

1 Integrating Databases and Programming Languages

The integration of databases and programming langauges is being motivated from two directions. The database community requires a more flexible and powerful way of modeling the world, whereas the programming language community wants the convenience of a reliable, efficient means of enabling entities to persist between program invocations.

1.1 The Programming Language Perspective

From the programmer's perspective, implementing the functionality of a database is no small task. Most programming languages provide some mechanism for recording information to disk at the granularity of a file. Without a database, much time and effort would be spent within applications converting structured entities of the language into a flat, unstructured format suitable for file transfer. This is further complicated by the presence of implicit object identity, as pointers between objects must be translated into explicit ids and recorded so that the corresponding links can be rebuilt.

But a database provides more than just a means of efficiently storing and retrieving large volumes of data. It allows concurrent access to data by multiple programs and users, while guaranteeing data integrity, (i.e. protecting the data from invalid updates via atomic transactions). To provide such features without a database would require the programmer to use the underlying file system as a locking mechanism. The application programmer would need to be concerned with the details of physical storage, such as caching disk updates, maintaining low-level data structures, and supporting recoverability in the face of hardware failure. Furthermore, a database provides a high level interactive query language, which allows simple queries to be expressed elegantly, and enables the use of indexing capabilities to improve response time for frequently accessed fields.

1.2 The Database Perspective

Traditionally, the query facilities provided to database users are not computationally complete languages, precluding arbitrarily complex processing of

data. Furthermore, they support only primitive data types, such as strings, floats, and integers, which makes it inappropriate for modeling certain real world applications.[1] The reason for such limitations is that they enable the system to perform automatic query optimizations which would not be possible in a more general language. However, if the database user required more complex processing and modeling, the use of an external programming language was necessary.

Some systems allowed the user to embed query processing facilities in a host language by linking with library modules, and calls to the database were treated like remote procedure calls, where the data was sequentially marshalled from the database into atomic variables via parameter passing, and those data types were then converted into the arbitrary data structures preferred by the language. This was generally inefficient, and lead to a bottleneck, in which the program is constricted because it has to force it's data into primitive formats which are perhaps unnatural. This became known as the *impedence mismatch problem*, which was analogous to the problem faced by programmers trying to convert their data structures into a format suitable for storage in a flat-file system.

2 Overcoming the Impedence Mismatch Problem

One approach for overcoming the impedence mismatch problem is to extend an existing programming languages with the notion of persistence, enabling it

[1] The earliest databases were hierarchical, in which the relationships between data objects were represented by pointers to disk locations, and the user was required to explictly traverse access paths through the physical medium. Because the physical details of the database were not hidden from the user, changing the structure of the data (i.e. adding a field to a record) was difficult, since the corresponding access paths had to be updated by the user.

In response to this problem, a new class of databases emerged: the *relational* database. In relational systems, the physical details are abstracted away from the user, and replaced by a logical view of the data. Data is organized into records, which are composed of logically related data items. Each record (or "tuple") consists of a group of named fields, each of which conforms to a basic data type, such as integer, float, or string. Records were grouped together to form tables of data which could be retrieved by the user through a standard relational query language.

to seamlessly interact with an object manager.[2] In such a system, persistent structures can be directly manipulated by the language, and facilities are provided to support relational query techniques over sets of data.

2.1 Milestones in Persistent Programming Langauge Development

The integration of database management facilities with a programming language was pioneered in **Pascal/R** [22], an extension of Pascal designed and implemented in the mid 1970's. The success of the Pascal/R system demonstrated that it was possible to extend the type system of a language to accomodate the relational model. Database schema were represented by a special *tuples* type, whose instances were able to persist beyond the lifetime of a program.

The first language to demonstrate uniform persistence was **PS-algol**.[4]. Whereas persistence in Pascal/R was limited to a subset of the types expressible in that language, PS-algol allowed all types be declared persistent. The notion of persistence being orthogonal to type has become an important feature of modern systems, allowing programmers to apply pre-existing code to persistent entities with little or no modification.

The first object-oriented langauge to be extended with persistence was Smalltalk in the early 1980's, which lead to the Gemstone system [16]. The software engineering benefits of object-oriented databases make them commercially viable for such CAD tools, office information systems, graphical information systems, and image processing applications. Soon, numerous implementations of persistent object-oriented languages began to arise, most of which were an extension to C++. These systems were mostly experimental, and varied greatly in their handling of persistence, due to the lack of a formal model for object-oriented databases. The next section discusses some emergent properties of these systems.

[2]Another approach was to extend the existing relational query language with new functionality to increase their power and flexibility. Some query languages, such as TEDM, were enriched by allowing declarative Prolog-style clauses, such as TEDM. Others systems were given object-oriented properties, such as Iris's Object-SQL, or extensions to INGRES' QUEL.

3 Comparison to Relational Databases

Traditional relational database applications were characterized by a slow, constant rate of data growth, and infrequent, centralized updates to database schema. But object-oriented databases tend to involve a much more sporadic pattern of data creation and schema design. Changes to schema are frequent and made by multiple users instead of a central database administrator.

Traditional databases allow a small number of atomic, fixed-size data types, with a large number of instances of each type. In contrast, object-oriented databases contain an arbitrarily large number of types which are extremely varied in size and structure (e.g. documents, images, etc.), and there may be only a few instances of each type. The relationships between types in a traditional database are mostly simple and fixed, whereas object-oriented applications are characterized by complex inter-object relationships which are subject to frequent change.

In relational systems, transactions are fine-grained and short-lived, supporting strong consistency during concurrent access. However, object-oriented database applications such as CAD development require transactions to be long-lived and may involve large portions of the database simultaneously. Such systems must have a relaxed view of data consistency in order to support concurrent access.

Relational systems are *value oriented*, in which relationships between records are established by having attributes with common values. Object-oriented langauges are *reference oriented*, in which relationships are established by embedding the identity of one object within another.

Data access in a relational system consists of associative joins within a set-based, closed algebra. Queries do not involve user-defined operations, and optimization techniques are relatively easy to implement. Object-oriented applications could benefit from such a query mechanism, but it is more natural to access data by *navigation*, i.e. traversing a path of pointers between objects.

4 Case Study: Two Object-Oriented Databases

In order to illustrate some of the issues that arise in object-oriented databases, we will compare the features of two existing systems: **ODE**, a prototype

system designed at AT&T, and **ObjectStore**, a commercial product from ObjectDesign. Both are persistent extensions to C++. Both meet all the requirements described in the previous section, providing facilities for creating and manipulating peristent objects, versioning, and associating constraints and triggers with objects.

4.1 Transparent Persistence

One of the main goals of Objectstore was to provide a unified programmatic interface, in which persistence is truly orthogonal to type. From the programmer's perspective, there is no difference between persistent and volatile objects. Dereferencing pointers is syntactically the same in both cases. Variables do not need to have their type declarations changed when persistent objects are used. Thus, C++ code written for transient objects can be used directly on persistent objects with little or no modification. A pointer can refer to persistent or transient data at different times during program execution. Furthermore, the access speed for persistent objects is usually equal to that of dereferencing transient data already in memory.

By contrast, ODE programmers must treat persistent objects differently from transient objects. There are three types of pointers in ODE: volatile pointers, persistent pointers, and dual pointers. Volatile pointers can only refer to volatile objects. Persistent pointers can only refer to persistent objects. A new persistent object is allocated by calling the special constructor pnew, which generates a unique object id that can be stored in a persistent pointer:

```
persistent Person *p;
p = pnew Person(``Jon Smith'');
```

The third pointer type, dual pointers, can refer to either volatile or persistent objects, and the question of persistence is determined at runtime. Unlike Objectstore pointers, a dual pointer in ODE cannot refer to both volatile and peristent objects within the same program; once it is assinged to a persistent object, it can only refer to persistent objects, and vice-versa.

The ODE designers rejected the Objectstore approach to persistent pointers beause they wanted an efficient implementation which would not require special hardware assists. By having no restrictions on pointers, a run-time check would be required on every pointer dereference. If a pointer happened

to point to a persistent object, it's address must be translated by the object manager before it is accessed. This check would impose unacceptable overhead on dereferencing non-persistent pointers, especially considering the heavy use of pointers in C++.

Objectstore avoids this runtime overhead by overloading the operating system's underlying page-fault mechanism. The addresses of persistent objects are denoted with illegal values (e.g. negative numbers), so references to them automatically incur segment violations. The object manager intercepts the segment violations and performs the appropriate translation into persistent store.

Although this technique combines the advantages of transparent persistence and fast access, it suffers from four problems:

1. In many operating systems, applications are allowed to intercept the segment violation signals. Such actions may inadvertantly interfere with the object manager, rendering persistent objects inaccessible. I encountered this problem while trying to build an application using Objectstore, and it was an extremely difficult bug to identify.

2. A user program may erroneously generate an illegal address which the object manager may interpret as a valid persistent address. Thus, the robustness of the system has been compromised.

3. Pointers to persistent objects are now limited to the size of ordinary pointers. This may be a problem for larger databases, which may require more address space than can be represented by a normal sized pointer. Also, it reduces the flexibility of what information may be stored as part of a persistant object's id.

4. Trapping mechanisms are architecture and operating-system dependent, and there is no machine-independent way to generate an illegal address, so the database system will be hard to port across systems.

By allowing the user to declare pointers to be of one type or another, persistent references can be handled at compile-time, eliminating all four problems. The ODE designers also claim that their pointer scheme leads to better program readability and allows better error checking by the compiler. In my experience with both systems, however, I am convinced that the

convenience of a single unified pointer mechanism outweighs the problems described by ODE.

4.2 Declarative Query Facility

One of the requirements of a persistent programming language is to provide a high level query language. The query language must be declarative, allowing relavitely complex data structures to be queried simply, and should provide obvious handles for query optimization.

The ODE system provides a declarative front-end language called CQL++ which is based on a type-generic object algebra that preserves the closure property of standard SQL. CQL++ allows users to directly manipulate objects without explicit use of their object ids. The encapsulation properties of the O++ versions of the objects are preserved, i.e. CQL++ users are not given access to private methods and data members. There are 3 views of an object: the CQL++ view, the O++ definition, and the object manager's view, which consists of an ¡object_id,state¿ pair[21]. The state corresponds to the private and public data members of the O++ object. The O++ object also consists of computed attributes, update methods, and other methods (non-updates). From the CQL++ user's perspective, the object merely consists of a set of query attributes (formed by the public data members and computed attributes), and the update methods.

An example of a CQL++ query is:

```
SELECT E.name, E.age(), E.dept.name
FROM Employee E
WHERE E.salary > 50000
```

In a relational database, the query would return a table consisting of three columns corresponding to the three selection attributes. In CQL++, SELECT creates a new *anonymous* object type which contains data members corresponding to the selection attributes. In this case, the anonymous object type has data members extracted from both the Employee class and the Department class. Although the attribute E.age() was originally a method, it is represented as a simple data member in the anonymous class.

New objects can be created via the INSERT statement. The following statement creates a new Course object by invoking the Course constructor.

The set of Students in the Course object will be initialized to contain all Student objects whose year data member contains the string 'fistyear':

```
INSERT INTO Course
VALUES ('CSE', 505,
        SELECT * FROM Professors WHERE name = 'Borning',
        SELECT * FROM Students WHERE student.year = 'firstyear')
```

CQL++ also provides DELETE and UPDATE statements. The UPDATE statement allows the user to set public data fields directly, or invoke public member functions for updating the object.

4.3 Collections and Iterators

In addition to CQL++, ODE also allows the expression of recursive queries through set iterators. In O++, sets are part of the language, and built-in operators allow assignment, union, difference, intersection, deletion, and set iteration. The iteration facilities provide an alternative to using object-ids for navigation. An arbitrary join can be expressed by iterating over clusters,[3] with join conditions specified as iteration conditions. This is achieved with the *for* and *suchthat* clauses, inspired by SQL. [4] Iterators can be nested, as demonstrated by the following O++ statement for retrieving a set of all employees who make high salaries and work in the same department as Alex:

```
persistent Employee *overpaid<>;
for p in Employees
  for q in p->dept->emps
    suchthat (p->salary > 50000 && q->name == 'Alex')
      overpaid->insert(p);
```

[3]All persistent objects of the same class are grouped together on disk in a *cluster* named after the class. Thus, clusters are type extents. The definition of new classes causes the object manager to automatically create a cluster for that type, and whenever a new instance is created, a pointer to it is inserted in the appropriate cluster. The programmer is freed from explicitly specifying and maintaining a class extent.

[4]The *for* statement applies only to objects which directly belong to the specified class, and not objects of descendent classes. To facilitate iteration over all instances of a class hierarchy, O++ provides the *forall* statement.

Objectstore provides a similar mechanism with a slightly different syntax. The above query would be expressed as:

```
os_Set<Employee*>overpaid =
all_employees
[: salary > 50000 &&‘
   dept->emps[:
     name == 'Fred' :] :]
```

Note that this is really a relational join. There are two paths along which to perform the query. One path is to iterate over the each employee in the employees set: for each employee that is overpaid, the query solver traverses that employee's dept pointer to see if the department contains Fred. The second path starts with Fred, traverses his dept pointer and then iterates over all the employees in that department, checking their salaries.

The query optimizer must decide which path to navigate based on the cardinality of each set and the presence of indices over data members.

Index maintenance is difficult, since arbitrary updates may involve side-effects for updating index values. If Employee instances are frequently accessed via their salary, it would be desirable to have an index over that data member. But it would not be practical to incur a run-time check on every object update to determine if index maintenance is required. Objectstore solves this problem by requiring the programmer to use the *indexable* keyword to explicitly tag those data members which could potentially be used as index types, the appropriate handling can be set up at compile-time.

4.4 Constraints

Constraint facilities have been traditionally provided in databases to maintain consistency beyond what is typically expressible by the limited type systems typically available. In persistent programming languages, the functionality of constraints can be implemented by the programmer. However, a constraint facility would still be useful, enabling the programmer to specify constraint relations declaratively rather than procedurally.

The database community and the programming language community have two different interpretations of the notion "constraint maintenance". In a constraint-based programming language, constraint maintenance involves

a constraint solver which resolves constraint violations via perterbation or refinement of underconstrained values. In traditional database systems, the notion of constraint maintenance is much more primitive: if a constraint is violated, simply abort the transaction and roll back any updates that have occured. Constraint maintenance in database systems has typically been limited to constraint *detection*.

ODE provides the latter form of constraint maintenence. The O++ language allows users to annotate class definitions with constraints (arbitrary boolean expressions) over their data items. Two kinds of constraints are provided: deferred and immediate.[5]n ODE terminology, a deferred constraint is called *soft*, and an immediate constraint is called *hard*. In a constraint-based programming language, hard and soft refer to the level of compliance expected of a constraint rather than the granularity of enforcement. To avoid confusion, I have chosen to rename the ODE constructs for this discussion. Immediate constraints are checked every time an object is updated. Deferred constraints are checked only at the end of a transaction. In database systems, a transaction is often the smallest unit across which integrity must be maintained. Finer granularity can be problematic, forcing an overly strict ordering among events within a transaction. Consider, for example, a checkbook balancing application in which a the balance is constrained never to fall below zero. In a single transaction, the user may write some checks and make a deposit. In this case, a deferred transaction is appropriate, since we are only concerned with the balance at commit time.

Another reason for having deferred constraints is to allow complementary constraint relationships to be maintained. Consider the classic example in which a person's spouse must have that person as a spouse:

```
class person
{
  ...
    persistent person *spouse;
    constraint:
      (spouse ==  NULL) || (this == spouse->spouse);
 };
```

Without deferred constraints, it would be impossible to record a marriage,

[5]I

since as soon as we tried to update one of the newlyweds, the constraint would be violated.

Objectstore provides a different solution for maintaining complementary relationships. The Objectstore language allows programmers to declare *inverse pointers*. If one side of the inverse pointer is set or deleted, the other side is automatically updated. This is a powerful mechanism for maintaining referential integrity, freeing the programmer from having to implement reference counting and garbage collection techniques. Inverse pointers are fully integrated with the collection facilities, and can be one-to-one, one-to-many, or many-to-many. Here is an example of a one-to-many relationship:

```
class Student
{
  public:
    ...
    Department* dept
       inverse member Department::grads;
};

class Department
{
  public:
    os_Set<employee*> grads
       inverse member Student::dept;
}
```

Whenever a Student is inserted into a Department's set of grads, the Student is automatically updated to refer to the Department, and vice-versa. Similarly, when a Student is deleted from a Department's set of grads, the dept pointer in the Student is set to null. If alex is currently in cse_dept, and alex-¿dept is reset to point at history_dept, then alex is removed from cse_dept-¿students and put in history_dept-¿students automatically.

In both ODE and Objectstore, constraints obey the inheritence mechanism. Derived classes inherit the constraints of parent classes and can be specialized with new constraints. Hierarchical constraints can be useful in many applications, such as frame-based knowledge bases.

I plan on extending this paper, and this appendix is a rough draft of future sections. Comments and suggestions appreciated!

A Active Databases: Triggers

In traditional passive systems, software components that update objects must themselves recognize the consequences of the update and then perform the appropriate actions. This compromises s/w modularity, since the updating component is logically performing the task of another s/w component that should respond to the update. Modularity can be maintained by having a separate s/w module that polls the objects and takes appropriate actions when they are changed in specified ways. but this wastes resources, and response time depends on polling period.

Passive systems are program-centric: applications must themselves check object states and initiate actions if conditions are satisfied. active db is data-centric. triggers automatically, applications need not worry about it. Actions can be notified by trigger if appropriate.

triggers. sometimes called "alerters" or "monitors". like constraints, monitor db for some conditions, except these conditions do not represent consistency violations. when trigger cond becomes true, trigger action is executed. active database.

A.1 Once-Only Triggers

triggers associated with objects. two types. *once-only* (default), and *perpetual*. once-only deactivated after it fires. can be reactivated explicitly. triggers specified within class defns:

```
trigger:
    [perpetual] T1 (params) : body
trigger body:
    trigg-cond ==> trigg-action

timed trigger:
    within expression ? trigger-cond ==> trigger-action
        [: timeout-action ]
```

timed trigger must be fired within specified period, or timeout action is fired.

to explicitly activate:

```
object\_id->T1(args)
```

when condition becomes true, trigger "fires", i.e. action becomes scheduled for action. independent transaction is created with trigger in body. trigger conditions are evaluated at the end of each transaction. trigger transaction weakly coupled to trigger action, i.e. occurs sometime after firing, but not necessarily imediately after.

trigger may explicitly be deactivated before fired:

```
~object-id->T1
```

special macro for use in trigger conds:

```
changed arg
```

true if arg has changed in current transaction, false otherwise.

A.2 Perpetual Triggers

perpetual triggers cannot be simulated with once-only triggers by reactivating trigger after trigger action. trigger action executed at some later time, so trigger will be inactive for the period until activation instruction in trigger action occurs.

```
class stock {
public:
  double price;
  ...
trigger:
  sellorder(int amount, double lowlimit, double time):
    within time ? price > lowlimit ==>
      sellstock(this, amount, price);
};
```

A.3 Intra-Object vs. Inter-Object Triggers

ODE uses intra-object triggers. intra = trigger associated with a particular object, and condition is evaluated when the obj is updated. inter = condition evaluated for multiple objects. for inter, would need to list trigger

in definition of each obj involved. alternatively: use a mechanism based on "friend". physical locality makes intra-object constraints and triggers more efficient. when event occurs, object being updated is already in memory. [11] says that "most inter-object constraints and triggers can be implemented using one or more intra-object constraint or triggers...A typical case is the "employee's salary no greater than the manager's salary" example. This is clearly an inter-object constraint...this inter-object constraint is then converted into two complementary intra-object constraints, one to be associated withthe employee and the other to be associated with he manager." p.14 [double-dispatching?]

[multiple dispatching! multi-constraints]

A.4 Eager vs. Lazy Computation

eager or lazy computation? suppose every object has attribute age() which may change on every clock update. eager method inefficient: compute new age for every person on every clock update. but attribute which is referenced often and updated infrequently should use pre-computation, i.e. eager. reasonable system should provide boh options. how to choose which one? POSTGRES optimizer decides. for ODE, virtual attributes are used to specify computation on demand, and triggers are used to specify computaion on update.

A.5 Triggers and Constraints in Other Systems

trigger and constraint mechanisms related, because constraints can be implemented as triggers. seperate facilities provided because they are logically different: constraints are used to ensure consistency: violation causes violating transaction to be aborted. in contrast, triggers are not concerned with consistency, they are initiated as seperate transactions. all objects of same type have the same constraints, but different triggers may be active or de-active for objects of same type.

facilities for active dbs first appeared in CODASYL: ON conditions.

System R provided triggers and constraints as mechanism for enforcing integrity.

Sybase provides facilities to specify RULES and TRIGGERS. RULES are integrity constraints that go beyond column datatype with no actions asso-

ciated. TRIGGER is a special kind of stored procedure that goes into effect when specified table is updated. roll back changes that violate constraint. can only be affected by changes to only one table. at most 3 triggers per table: update, insert, delete trig. one trig cannot call another.

these limits not in ode.

in many systems, integrity constraints and triggers are expressed in a seperate lang that is distinct from normal db query lang. difficult to invoke arbitrary actions when such triggers and constraints fire. difficult to check conditions on every update.

B Misc. Features of Object-Oriented Database Systems

field of oodbs is currently getting lots of attention from both experimental and theoretical standpoint. field is characterized by lack of common data model, lack of formal foundations, and strong experimental activity. clear specification exists for rdb data model and query langauge, but not for oodbs. no consensus on data model.

logic programming has strong theoretical framework, but not oodbs. thus ill-defined semantics of such concepts as types.

characteristics it should possess:

B.1 Mandatory Features

[9]

support complex objects. minimal set of constructors: sets, tuples, lists. sets for representing collections, tuples for representing properties, lists to capture order. all 3 things occur in real world.

these 3 constructors should appy to any object, i.e. be orthogonal to type of obj. relational model constructors don't meet this requirement, since set constructor can only be applied to tuple, and tuple can only be applied to atomic values.

object identity. object sharing (i.e. components can be shared, not just copied). supporting identity implies offering object assignment, deep and shallow copy, tests for identity and equality (both deep and shallow). new concept for pure relational systems, where relations are value based.

encapsulation. need for distinction between implementation and specification, and need for modularity.

types or classes must be supported.

class or type hierarchies shall be supported.

computational completeness shal be provided. more than SQL.

extensibility. no distinction between system-defined and user-defined types from the application programmer's perspective.

persistence. user should not have to explictly copy or move data to make it persistent.

secondary storage management. must be able to handle very large amounts of data. performance features: index management, data clustering, data buffering, access-path selection, query optimization. should be invisible to user. [gripe with ode].

concurrency. ensure harmonious coexistence among users working simultaneously. controlled sharing, atomicity, some degree of serializability.

recovery from h/w and s/w failures. bring back to some coherent state of data.

ad hoc query facility. simple queries should be simple. relational queries should be supported. 3 criteria: high level and declarative, efficient (lend itself to optimization), and application independent.

B.2 Optional Features

- multiple inheritance.
- type checking and type inferencing. more compile time checking, the better.
- distribution.
- design transactions. ie. long, nested transactions.
- versions.

B.3 Swizzling

PS-Algol uses pointer swizzling to optimize access to persistent objects. first reference to persistent object results in its being cached in volatile memory

and original pointer is replaced by ptr to cached location. when obj is written back to store, swizzled ptrs that contain the cache address of this obj must be deswizzled.

B.4 Referential Integrity

if entity is referred to in some relation, then the entity must exist in a "primary" relation listing all entities of that type. e.g., if "Supplier" relation records a company as supplying a certain part number, then the part number must be a valid part number recorded in some "Parts" relation. In an object-oriented db framework, referential integrity means that htere should not be any refs to non-existent objects. db must guarantee that if an object id is recorded in db, corresponding obj must be present in the db.

when object is deleted, need reference count or indexing to enforce referential integrity.

B.5 Versioning

representation of multiple object versions is essential to many applications, particularly engineering design activities. 3 approaches:

1. versioning is so fundamental that language primitives should be provided for versioning. ie. part of data model. example is DAMOKLES [8], which allows classes to be versionable.

2. versions not considered part of data model, but a version service is provided by s/w layer implemented directly on top of kernel data model. Encore [23] implements versioning as a seperate layer on top of basic data model.

3. versions considered to be application issue. apps must set up their own types for recording version control. GemStone [16] takes this approach. Create a version class, which has info such as "version-number" and "latest update", whose behavior can be inherited by sublcasses.

versioning in ODE: all persistent objects can have versions. no predefined limit on number of versions object can have. can access latest version or specific historical version. object and all its versions considered one logical

object with one id. ptr can refer to a specific version of an object. need to explicitly create a new version.

```
newversion(p)
```

henceforth, p refers to new version. access nth previous version of p:

```
pp = previous(p,n)
```

given p, pdelete deletes object and all versions. given pp, pdelete only deletes version.

C Other Object-Oriented Systems

C.1 BETA

BETA [17]

when an object is made persistent, all objects that can be reached via references will also be persistent. this includes statically enclosing objects. transitive closure of the object.

persistent store is itself a BETA object. = collection of persistent objs. current implementation: obj in one pers.store cannot refer to obj in another.

persistent root = entry point to store. has string name.

implementation issue:

```
(# T: (# c: @integer;
         A: (# b: @integer do c->b #);
         X: @ A;
         Y: @ A;
      #);
   V: @ T;
   W: @ T
#)
```

V and W are instances of T. Each has internal attribs. The pattern attribute A of V is different from W.A, since each may only refer to the attribs of their enclosing object V or W.

Betaenv is a fragment defining a pattern that encloses all BETA code. each program creates a new betaenv object. patterns described in different

programs will never be identical, since they will always be attribs of the betaenv instance created by program execution.

thus, patterns described in different executions will never be identical. to overcome this prob, betaenv is made to be a persistent object existing on the entire system/network where persistent objs are to be shared, so all progs will have the same betaenv.

implication: if multiple progs want to share a pattern, it should be defined in a library slot if it wil be used to generate persistent objects.

object has a reference to it's corresponding fragment and compiled code. if the frag changes, the corresponding persistent obj becomes obsolete. current implementation does not check integrity of objs, i.e. it is up to the user to make sure frag hasn't changed since object's creation.

[maybe use texthashtable example pp.6-7?]

object can be saved as a persistent root via:

(R[],'foo') -¿ PS.put;

PS is an instace of persistentStore. All objects referred by R are also made persistent.

object is retrieved by:

```
('foo',T##) -> PS.get -> R[]
```

where T## is a pattern from a library.

if an object is modified after having been sved, the state of the modified object is not automatically saved. a new put operation must be executed.

actual storing of persistent objects occurs when operation "close" is executed on persistent store.

pesistent store is stored as a file on secondary storage.

C.2 PROCOL

PROCOL[13]

object-oriented language given persistence. major motivation: GIS, geometric and graphic data.

upward compatible: existing non-persistent procol programs can be compiled by procol compiler.

transparent persistent objects: persistence independence (i.e. persistent and volatile objects interchangeable as parameters), and persistence data

type orthogonality: all objects allowed to be persistent, no matter how complex the type.

extendibility with new ADTs: as compared to traditional dbms's, which allow only basic types.

efficient associative searching (via B-trees, etc.).

highly interactive for graphical applications.

D Constraint-Based Imperative Languages

D.1 Constraints and Object Identity

From [14]: constraint imperative programming = integrate declarative constraints and imperative oo programming. goal: use constraints to express relations among objects explicitly instead of implicitly. but object-identity can result in implicit relations. solution: identity constraints.

CIP Identity Gap

D.2 Kaleidoscope

From [10]:

constraints useful because they are declarative: emphasize relation itself instead of procedural steps necessary to maintain the relation. obstacle to thorough integration of constraints in an object-language: objects provide a larger, arbitrarily general computational domain, which cannot be solved by an efficient constraint solver.

like kaleidoscope, ode's constraints obey the languages inheritance mechanism

kaleid. has required and preferential constraints. constraint hierarchy: arbitrary number of levels of preference.

D.3 Adding Transactions to Kaleidoscope

when are they solved? is the solver automatically invoked whenever a variable changes, or does the programmer have to explictly trigger the solver when necessary?

Motivation for transactional approach:

"The Kaleidoscope'91 language and implementation meets five of our six original goals...The one goal that it does not yet meet is number 6: reasonable efficiency...To this end, our implementation and research effort is divided between improving our constraint solvers and developing specialized compilation techniques."

[Do we really need the solver to be invoked on every update? If programmer had explicitly control as to when, we have ODE deferred constraints, i.e. transactional. could be more efficient, and solve the following:]

D.4 Techniques for Integration: Summary

local propagation: start with known set of values and use constraint to determine other values. simple and fast. but limited to acyclic constraints, and no partial info constraints (i.e. ¿).

constraints on Primitive Leaves: seperate the domain into user-defined objects and primitives, split compound constraint into basic ones. bad because it freezes variable types, and violates encapsulation when asserting inter-object constraints, since leaves must be accessed directly and they may be private instance variables.

define new solver to handle new domain.

kaleidoscope uses multiple dispatching: implementation inheritence graph of EACH arg is searched for applicable methods, which are placed into a parital order.

conflict between declarative constraints and imperative assigments, which are destructive. assignment is defined as an equality constraint between the PREVIOUS value of the RIGHT expression and the CURRENT value of the LEFT variable. Thus, all expressions denote constraints.

all expressions are defined relative to a CURRENT TIME. current time is advanced at the end of each statement. compound statements can be created without advancing the clock by using —— operator.

constraints can be once, always, assert during (loop).

constraint constructors are dispatched upon. they are free of nonlocal side-effects. may not contain out-of-scope variables or advance the global time.

"greatest savings in execution time will come from moving as much of the constraint satisfaction problem from run time to compile time as possible." p.12.

E Persistent Prolog: Motivation and Issues

- Language Perspective: Prolog could use the ability to treat large external databases as though their data was inside prolog. Tight coupling to database system.
- Database Perspective: prolog = powerful extension to query language. Allows tools to be written which aid expert in rule definition. Rules and facts = flexible, elegant query language
- recusive rules
- metalogical features: alter how query is solved, or have side-effects which may alter results of subsequent queries.
  - retract- remove fact from db
  - asserta, assertz
  - cut- restrict backtracking options

E.1 Expert Systems

PROLOG and EXPERT SYSTEMS /citelucas-88 prolog good for expert systems because it is:

- rule-based:
  - rules largely independent of eachother.
  - naturally models human skills, which are rule-based.
  - rules can be incrementally updated.
- declarative: rules easily inspected. but programmer should be aware of the procedural implications of the rules he writes in order to write more efficient programs, since careless ordering of subgoals can cause much unnecessary searching.
- facilitates programs that can explain their own reasoning: intermix rule bodies with text statements to trace subgoal fulfillment.

E.2 Why Prolog Could Use Persistence

- limited, unstructured internal database:
  - partial solution: link to external filesystem.
  - linkage should be transparent.
  - tightly coupled: each record processed before subsequent record is retrieved.
  - loosely coupled: table downloaded into prolog, then records processed one at a time.
- motivation: existing relational db users can access their data via logic language without the necessity for preprocessing. powerful pattern-matching and reasoning capabilities of prolog make it a good tool for implementing high-level front-ends. i.e. use it to apply expert-system rules to analyze db data. facilitates recursive queries.

most prolog systems represent the set of clauses in primary memory, either as literal representatinos of prolog source clause (in interpretive systems), or as executable machine code (in compiled systems). prolog internal database organized as a heap, with clauses dynamically allocated and deleted as records within a network of linked lists. degrades when number of clauses is more than a few thousand. /citeirving-88.

constraint on max size of system: clauses held in main mem. many prolog systems use virtual mem techniques to page sections of db to and from backing storage. paging is random, leads to poor performance. procedural languages exhibit locality of reference, but not prolog.

clause indexing: prolog systems use hashing techniques to identify the set of clauses for the relation corresponding to the current goal. hashing mechanism is extended with predicates so that goal args can be used to restrict subset of candidates which are tried sequentially via unification and backtracking. works well for static clauses representing procedural part of prog. however, does not allow indexed predicates to be modified by the addition or deletion of component cluases, as is often the case with declarative predicates representing dynamic info. we need more general clause indexing scheme which allows dynamic modification of clause store without extensive recompilation and rehashing. B-trees are a good mechanism.

to solve real world problems, we want to use prolog reasoning on pre-existing non-prolog data, which may exist in multi-tasking environment, which requires dbms support.

E.3 Database Interface for Prolog

minimum specification for interface:

- generality of representation. must support transparent mapping of complex tree-like prolog terms into linear db records.
- uniform syntax and transparent unification: ideal situation: programmer specifies each table he intends using and what it is to be known by. from that point on, use of table exactly the same as using sets of facts in prolog's internal db.
- interface should determine column names of tables and know how to tranlate between param position and col name. thus, interface should access data dictionary.
- transparent backtracking: searches on tables must mimic prolog's search mechanism. left to right, depth first, allow recursion. need to interleave searches on different tables and allow multiple simultaneous searches on the same table. We need: CURSORS. cursor = control block defining current status of search (position within table).
- low performance overhead required. sequential clause lookup will be much slower than for internal clauses. but indexed access should be acceptably fast using external dbms. as much as possible, mapping should be copmiled and not interpreted at runtime.

E.4 Translation Between SQL and Prolog

2-D tables of records. set of facts all with same functor and same number of components. unlike prolog, columns of tables have LABELS. in prolog, field position replaces names of columns. select conditions are set via variable instantiation. [show prolog facts and table]

gateway = inter-query-language translation between dbs.

[18] describes how to implement SQL and QBE queries in prolog.

each relation comprises a set oftuples. each tuple can be represented as a ground unit clause in prolog:

```
emp(1,'fernando'2,30,3).
emp(2,'carlos',3,60,3).
emp(3,'jose',1,25,4).
```

For each relation, there is a scheme clause which defines the structure of the relation: name, no.tuples, no.attribs, list of attrib names.

```
scheme(emp,4,5,[empno,name,dno,sal,mgr]).
```

SQL processor translates query into prolog clauses:

```
SELECT name,sal
FROM emp
WHERE dno=2
```

becomes:

```
emp(_,X1,2,X3,_).
```

each attrib name that occurs in SQL query needs to be fully identified, checking the relation to which it belongs and its position in the list of attributes. each attrib name is mapped into a structure of the form: [Relname,Label,Position,Attname]. A special parser is used to build a structure representing the SQL query before translating it into prolog query.

[15]

equi-join: relates data between tables. i.e. here's a join on MANUF field

```
SELECT ADDRESS FROM SUPPLIERS
  WHERE CARS.MODEL = 'Metro' AND
  SUPPLIERS.MANUF = CARS.MANUF
```

here's the prolog version:

```
?-car(Man,metro,_,_,_,_,_),supplier(Man,Address,_),write(Address),nl.
```

key fields = fields for which an index will be created allowing direct access retrieval of entire record whenever key value is known. unique key can either be new field with unique id, or concatenation of several fields to define key.

E.5 Meta-Translation

source-to-source meta-translation system which generalizes the production of translators. uses intermediate relational algebra tree.

- source-input to tree translator: syntax of new query language is entered via meta-notation that allows query language semantics to be easily specified.
- tree to source-output translator: specified via form-based interface which prompts user for syntactic constructs in the new query language.

motivation: distributed, heterogenous db systems need to talk to eachother.

E.6 MIMER: A Back End to Prolog

Mimer = relational database used by [15] as back-end to prolog.

prolog search mechanism duplicated via: stack of cursors. new search on a table opens a new cursor, which is pushed onto stack whenever there remained furthre rows in the table that cold be retrieved via sleect condition. to continue search, pop cursor. selection conditions established via current variable instantiation set.

declare external table:

```
?-external(manufact,'MANUFACT').
```

Every time prolog processes goal which is named as the first param, it should access Mimer table given as second param. interpreter associates a flag with each functor: external functors contain pointers to Mimer table names.

accesstab: build template of next table entry, leaving blanks on unknown fields. pass template to fortran routine, which accesses db and fills in blanks.

existing prolog programs should work without modification, their datasets being arbitrarily extended via interface.

CUT must be supported: cut removes all backtrack points as far back as the parent goal from the backtrack stack. prolog must tell interface when a cut occurs, how many cursors to be popped and discarded.

ASSERT and RETRACT: must support dynamic modification for clauses held externally.

basic update primitives assert and retract should be encased in transaction primitives in order to ensure conistency in a multi-user environment. special problems arise with asserta and assertz, which specify WHERE facts are inserted. Most database applications do not provide this flexibity.

concurrent reading and writing must be supported: consider

```
?-p(X),assert(p(b)).
```

Call to p(X) does not complete before assert(p(b)) is called. Data structures associated with p must allow concurrent reading and writing. deadlock must be avoided...

CALL and NOT: solve routine called recursively for new goal sequence...maintains appropriate backtracking behavior.

need powerful garbage collection techniques

optimisation: consider prolog goal:

```
?- employees(Name,Salary,_,_), (Salary>10000),...
```

inefficient to retrieve each row and backtrack if salary condition not met. database software may have efficient index on salary key. database search notified of optimisable constraints.

=OTHER PROLOG PERSISTENT SYSTEMS

Another persistent prolog implementation: Perlog citemoffat-88, tightly coupled with PS_algol back end POMS. Rather than having a global name space, Perlog parititions name space with modules, which can be committed individually to backing store.

References

[1] R. Agrawal and N. Gehani. Ode: Object database & environment. *SIGMOD*, 1989.

[2] R. Agrawal and N. Gehani. Ode: Rationale for the design of persistence and query processing facilities in the database programming language o++. In *2nd International Workshop on Database Programming Languages*, Portland, Oregon, 1989.

[3] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 430–440, Orlando, October 1987.

[4] M. Atkinson and et. al. P. Bailey. An approach to persistent programming. *Computer Journal*, 26(4), November 1983.

[5] F. Bancilhon and P. Buneman. *Advances in Database Programing Languages*. ACM Press, New York, 1990.

[6] A. Brown. *Object-Oriented Databases: Applications in Software Engineering*. McGraw-Hill, Berkshire, England, 1991.

[7] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of SIGMOD 84*, volume 14, pages 316–325, June 1984.

[8] K. Dittrich, W. Gotthard, and P. Lockemann. Damokles: A database system for software engineering environments. In *Proceedings of the IFIP Workshop on Advanced Programming Environments*, Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science.

[9] C. Delobel F. Bancilhon and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, CA, 1992.

[10] B. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 268–286, 1992.

[11] N. Gehani and H. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB 1991*, 1992.

[12] T. Irving. A generalized interface between prolog and relational databases. In P. Gray and R. Lucas, editors, *Prolog and Databases: Implementations and New Directions*, pages 81–94. Ellis Horwood, Chichester, England, 1988.

[13] C. Laffra and P. Oosterom. Persistent graphical object. In *Advances in Object-Oriented Graphics I*, pages 95–129. EUROGRAPHICS Seminars, Germany, 1991.

[14] G. Lopez, B. Freeman-Benson, and A. Borning. Constraints and object identity. In *European Conference on Object-Oriented Programming*, 1994. Submitted.

[15] R. Lucas. *Database Applications Using Prolog.* Ellis Horwood, Chichester, England, 1988.

[16] D. Maier and J. Stein. Development of an object-oriented dbms. In *Proceedings of the 1986 Object-Oriented Programming Systems and Languages Conference*, pages 472–482, 1986.

[17] Persistence in beta. Technical Report MIA 91-20-0.3, Mjolner Informatics, 1992.

[18] F. Mouta, M. Williams, and J. Neves. Implementing query languages in prolog. In P. Gray and R. Lucas, editors, *Prolog and Databases: Implementations and New Directions*, pages 13–21. Ellis Horwood, Chichester, England, 1988.

[19] J.E. Richardson and M.J. Carey. Persistence in the e language: Issues and implementation. Sciences 791, Univ. Wisconsin, Madision, September 1988.

[20] L. Rowe and K. Shoens. Data abstraction, views and updates in rigel. In *Proceedings of ACM SIGMOD*, 1979.

[21] N. Gehani S. Dar and H. Jagadish. Cql++: A sql for a c++ based object-oriented dbms. In *EDBT 92*, Vienna, Austria, 1992.

[22] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–281, September 1977.

[23] A. Skarra and S. Zdonik. The management of changing types in an object-oriented database. *ACM SIGPLAN Notices*, 21(11):483–495, 1986.

[24] D. Stemple and T. Sheard. *Construction and Calculus of Types for Database Systems*, pages 3–22. ACM Press, New York, 1990.