XStore/J

User Guide

XAVAX, Inc.

Contents

CHAPTER 99 *Planned Enhancements*

Preface

XStore Goals

XStore/J is the Java variant of the XStore persistence framework. I began developing XStore with the goal of achieving the best of two worlds: the programming simplicity of an object-oriented database with the availability of a relational database. A programmer employing the object-oriented programming paradigm would rather deal with classes, fields, and associations rather than tables, columns, and relationships, and an object-oriented database supports this view of persistence. Unfortunately, object-oriented database products have not met with much acceptance in the mainstream IT community where relational databases are pervasive.

XStore attempts to bridge this gap by implementing the client-side features of an object-oriented database while using a relational database as a *storage engine*. From the programmer's perspective, XStore operates like an object-oriented database. Persistent objects are created, manipulated, and deleted much like transient objects with very little attention to the details of object persistence. Persistent objects automatically become part of the current transaction when they are created or loaded and any changes are automatically persisted when the transaction is committed.

Every attempt has been made to implement the relational side of XStore in a manner that would be palatable to most relational database administrators (DBAs). Each class is mapped to one or more tables and each field is mapped to a column of the most appropriate database type. This is preferable to the alternative of saving the entire object as a binary large object (BLOB) since existing relational tools can be used to examine objects in the database. Associ-

ations are modelled as relationships using foreign keys, in the case of 1:1 associations, and separate cross-reference tables in the case of 1:n associations. The 64-bit object identifier (OID) is mapped to a column, usually of type BigInt or Number(20), and is always used as the primary key of the table. The OID is best presented to DBAs as a *surrogate* primary key created by a sequence generator, as DBAs tend to favor surrogate keys.

Who Should Use XStore?

XStore differs from other persistence frameworks such as Hibernate or TOPLink in that it does not attempt to be a general purpose framework. XStore strives to achieve one goal: to emulate an object-oriented database in a relational world. Toward that purpose, XStore requires the user to accept common object-oriented database concepts such as a global object identifier (OID). If your application is required to support a legacy database schema or the schema is otherwise constrained in such a way that will not allow using the OID as the primary key, then XStore is not appropriate for your application. If your application's primary purpose is reporting and does not involve manipulating objects, then a relational view of the data is more appropriate and the overhead of XStore or any object-oriented database would result in lesser performance. However, if your goal is to use the object-oriented paradigm with minimal attention to the details of persistence, while living in a relational world, then I believe XStore is the best tool for the job.

Licensing

The use of XStore is governed by the Xavax Open Software License. To summarize, you can use XStore for any purpose, including commercial applications, without paying any license fees or royalties. You can distribute XStore provided you acknowledge the copyright. You can modify XStore and distribute modified copies provided that the modified copies are clearly marked as having been modified and the recipient is informed of how to obtain the original source. If you modify XStore, you are encouraged to submit those changes to the author who will review the changes and consider them for inclusion in future versions of XStore.

Required Software

XStore has been tested using JDK 1.4.2 and 1.5. It requires a JDBC driver that implements prepared statements, which is almost any JDBC driver available. XStore uses Apache Commons Logging so that library must be in the class path. XStore can operate in a container such as Weblogic or Tomcat and in that case it will use the standard facility for looking up data sources; however, a container is not required. When running standalone, XStore provides a set of utility classes which implement the lookup mechanism and the DataSource interface.

Web Support

The latest information about XStore including documentation of the XStore API is available at the XStore web site.

```
http://www.xavax.com/xstore/
```

The author can be contacted as follows.

```
mailto:alvitar@xavax.com
phone: +1.404.468.0626
http://www.xavax.com
```

Acknowledgements

I began implementing my first persistence framework in 1992. Faced with the prospect of writing CRUD methods for 52 classes, I thought “there has to be a better way” and began searching for a way to simplify the persistence of objects. I gleaned many good ideas from the writings of Scott Ambler (www.ambysoft.com) and his work had a significant influence on that first design and on XStore. I was familiar with the concept of metadata, such as provided in Objective/C, but this was the first time I had considered using metadata to facilitate the serialization or persistence of objects.

Since that time I have implemented five persistence frameworks (1 in C, 1 in C++, 3 in Java). One might contend that makes me an expert on how not to design a persistence framework. Along the way I had the opportunity to work with a few commercial object-oriented databases and XStore borrows liberally from the concepts learned during the journey. Especially worth noting is Versant. XStore’s LOID and object cache directory are similar to the LOID and Common Object Descriptor (COD) Table in Versant.

I would like to thank the good people of bioMerieux Vitek (St. Louis), IBM Mapping Applications Development (Kingston, NY), Idapta (Atlanta), and Scientific Technologies Corporation (Atlanta) for supporting my work on previous frameworks. I want to thank a former manager, Dr. Dibyendu Baksi, for encouraging and supporting my work and being a sounding board for my ideas. Last but certainly not least, I thank my family for supporting me and tolerating my almost constant use of the computer during my copious spare time.

Phillip L. Harbison, CTO
Xavax, Inc.

XStore was developed on an Apple Powerbook G4 using Eclipse and GNU Emacs. XStore documentation was prepared using Adobe Framemaker and Javadoc.

CHAPTER 1

Introduction to XStore/J Persistence Concepts

XStore is a framework for persisting objects in a database. From a programmer's perspective, XStore operates much like an object-oriented database (OODBMS); however, the persistent objects are transparently mapped to tables and columns and stored in a relational database (RDBMS). XStore/J (XStore for Java) supports any relational database that provides a JDBC driver supporting prepared statements.

Persistent objects in XStore are created and behave much the same as conventional transient objects. The changes necessary to make a class persistent are minimal. Unlike transient objects, persistent objects do not cease to exist when a program terminates; therefore, persistent objects must be deleted explicitly.

Operations on persistent objects occur within the context of a transaction. During a transaction, persistent objects are created or loaded into the object cache where they can be modified or deleted. If a transaction is committed, any changes made in the object cache are committed to the database. Objects created in the object cache are persistent after a commit. If a transaction is rolled back, modified objects are removed from the cache and will be reloaded automatically as they are used. Objects created in the object cache cease to exist after a rollback.

Persisted objects are located and loaded into memory in one of two ways. Finder methods are provided which locate and load objects which match a set of criteria provided by the programmer similar to an SQL where clause. Persistent objects associated with objects already in memory are automatically loaded when the first attempt is made to reference the object through the association.

Object Identity

Transient objects have an identity which is usually the address of the object in memory; however, this identity ceases to exist when the program terminates. In XStore, persistent objects also have a permanent identity known as a Logical Object Identifier (LOID). As shown in Figure 1, a LOID is composed of a 16-bit database identifier, a 16-bit class identifier, and a 64-bit object identifier (OID). The OID is generated automatically by XStore and is unique across all classes and databases in an XStore environment. A LOID contains all of the information needed by XStore to locate a persistent object and is capable of addressing 65,536 databases, 65,536 classes, and 18 quintillion[1] objects. An XStore installation creating one billion objects per second would require over 500 years to exhaust the set of possible OIDs.

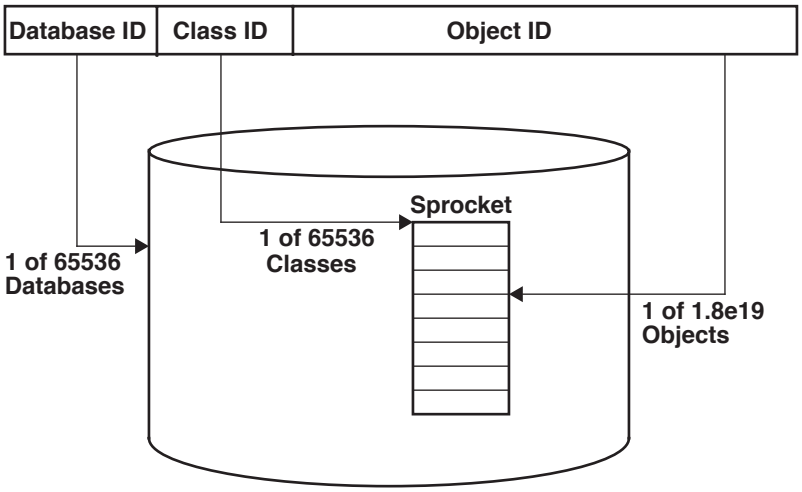

FIGURE 1. Logical Object Identifier

Proxy Objects

A *proxy* is a surrogate or placeholder for another object. XStore/J uses proxy objects to control access to an underlying *implementation object* residing in the object cache and to implement *demand loading*. When a program locates a persisted object using a query, the object returned to the program is actually a proxy for the underlying implementation object which may not yet

1. 18 quintillion objects is equal to 18 billion, billion objects.

be loaded into the object cache. When the program attempts to use the proxy, the underlying object is loaded automatically.

The proxy implements the same interface as the underlying object; therefore, it can be used in any context where the underlying object could be used. For example, if we have a class called WidgetProxy which is a class of proxies for objects which implement the Widget interface, then a WidgetProxy can be used anywhere a Widget could be used.

It is important that programmers only create proxy objects rather than implementation objects since the proxy is XStore's means for controlling a persistent object. To avoid the mistake of creating an implementation object, it is a good practice to always use factory classes to create persistent objects. XStore/J provides a tool that automatically generates proxy and factory classes containing constructors and create methods with the same set of parameters as the constructors for the implementation class.

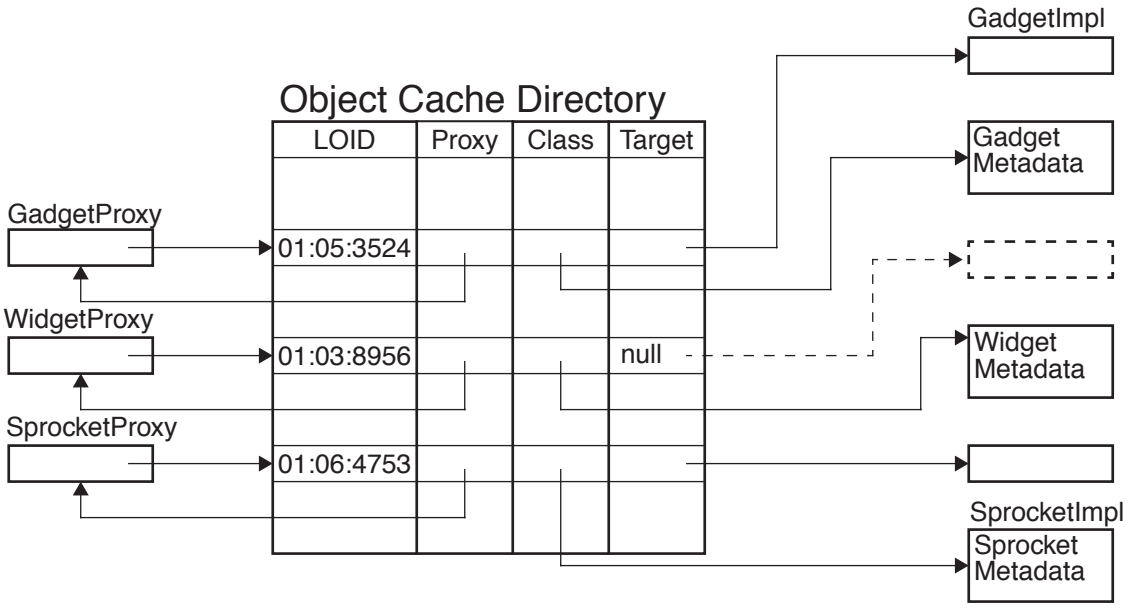

FIGURE 2. XStore Proxy Mechanism and Object Cache Directory

Object Cache

All active persistent objects reside in an object cache managed automatically by XStore. As shown in Figure 2, the *object cache directory* includes the LOID of the object, a reference to the proxy, a reference to the class metadata for the object, and a reference to the object which will be null until the object is loaded. There are also additional flags not shown which are used by XStore to track the state of the object. There are several advantages to using an object cache. The programmer does not have to be concerned with which objects are in memory since XStore manages the cache and loads objects on demand. If there are no applications which concurrently modify the database, the cache can improve performance of the application by keeping objects in memory after a transaction is committed. All information pertaining to the persistence of an object, such as the LOID, is stored in the object cache directory rather than in the object itself. This makes it possible to persist any object without changing the object.

Interface versus Implementation

For the proxy mechanism to work, the programmer must define interfaces and classes that implement the interfaces. To use our previous example, the developer might take an existing, non-persistent class such as Widget and rename it WidgetImpl (for *Widget Implementation*), then extract the interface which would be called Widget. XStore provides a tool to assist in interface extraction. Some IDEs also provide interface extraction tools.

Inheritance

Like any OODBMS, XStore supports inheritance and offers a range of solutions for persisting specialized classes. In the fully normalized form, a table would be created to store the properties of the base class and additional tables would be created to store the properties unique to each concrete class. In the denormalized form, a table would be created for each concrete class and would store all properties of that class including properties of any base classes. Figure 3 illustrates the two persistence strategies for an example application with a Vehicle base class and concrete classes for Airplane and Automobile.

The persistence strategy of a class is independent of the persistence strategy for all other classes. In this example, the developer could choose the fully normalized strategy for Automobile and the denormalized strategy for Airplane. There is one major disadvantage to using the denormalized strategy. If objects of any class derived from Vehicle such as Airplane are not represented in the common Vehicle table, then a search of the Vehicle table will not consider many possible matches such as airplanes that are vehicles. This limits the usefulness of polymorphism in an application. The disadvantage of the normalized strategy is it requires one or more joins to retrieve an object. XStore automatically performs the joins necessary to retrieve

entire objects, thereby avoiding the “object slicing” problem that often plagues many ad hoc persistence solutions.

An inheritance hierarchy often has multiple levels and XStore supports this. To expand on our previous example, Automobile could be converted to an abstract class called LandVehicle with concrete classes such as Automobile, Truck, and Tractor. There is no upper limit on the number of joins XStore will perform to accommodate a multi-level inheritance hierarchy; however, the number should be limited to minimize the performance impact. The programmer also has the flexibility of choosing to normalize at one inheritance level and denormalize at another level. In this example, there could be a shared Vehicle table, but Automobile, Truck, and Tractor could duplicate any properties common to any LandVehicle.

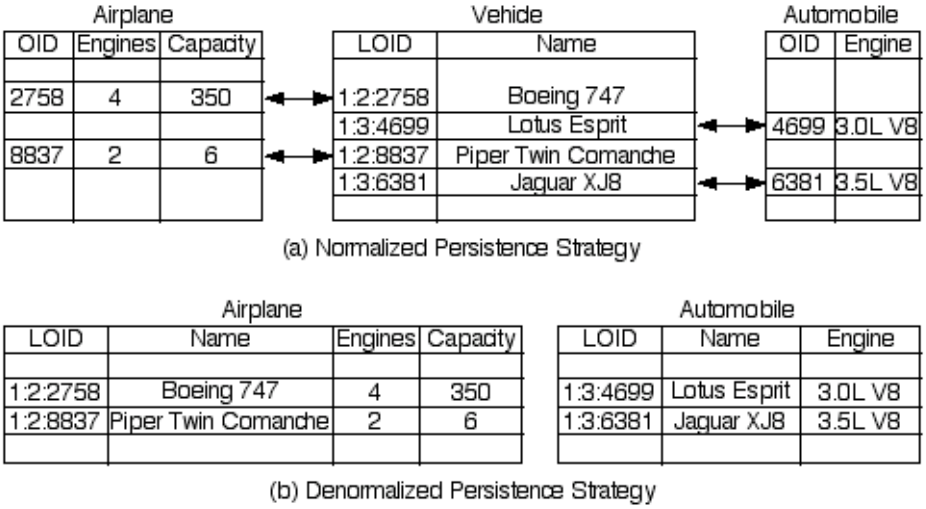

FIGURE 3. Persistence Strategies for Inheritance

Polymorphism

XStore’s demand loading and finder mechanisms support polymorphism. When searching for objects, the programmer specifies a class X and a set of criteria involving properties of X. If X is a base class for other persistent classes which use the normalized persistence strategy, then the find results may include instances of the derived classes. If a persistent object has an association to an object of class X, the object can belong to any class derived from X. When the associated object is loaded, XStore examines the class ID contained within the LOID and constructs a proxy of the appropriate type.

Associations

XStore supports 1:1 and 1:n associations to other persistent objects. When a persistent object is in memory, it contains a reference to the proxy for the associated object, or, in the case of a 1:n association, a collection of references to proxies. When the persistent object is stored in the database, the LOID of the associated object is stored along with the object. A separate table with the source LOID, destination LOID, and sequence number are used to persist 1:n associations. The sequence number is needed to make each row unique but also allows XStore to preserve order in collections of associations. Figure 4 illustrates how 1:1 and 1:n associations are persisted in a database.

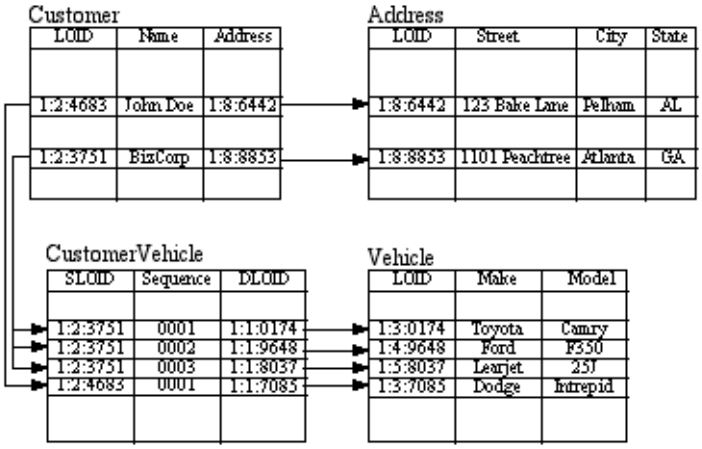

FIGURE 4. Persistence of 1:1 and 1:n Associations

Any class implementing the *Collection* interface can be used to store the proxies contained in a 1:n association. XStore examines the type of the class member used to store the collection to determine the appropriate container class. If the member is already initialized, XStore will reuse the container after calling the container's clear method to remove preexisting objects.

Implementations of the *Map* interface cannot be used directly to store a 1:n association since the object loader has no means of determining the key required by the put method; however, XStore provides a convenient solution. The *CollectionMap* interface extends *Collection* adding the Map methods not included in Collection (get, put, containsKey, makeKey, and removeKey). AbstractCollectionMap implements all of these except makeKey and delegates to an underlying map passed to it in the constructor. The programmer need only extend AbstractCollectionMap and implement the makeKey method to have a persistable map of associations. Here is

an example of a class that might be used in a real estate application to provide a 1:n association of Customer to Property with the ZIP code as the key to an underlying HashMap.

```
public class CustomerPropertyMap extends AbstractCollectionMap {
  public CustomerPropertyMap() {
    super(new HashMap());
  }
  public Object makeKey(Object o) {
    return ((Property) o).getZipCode();
  }
}
```

Mapping Metadata

Before a class of objects can be persisted, *metadata* must be provided which describes how the properties and associations of an object should be mapped to tables and columns. XStore provides a *metadata generator* tool which will examine a list of classes and automatically produce a list of SQL statements to load the metadata and database schema for persisting each class. The programmer may edit the metadata to specify field lengths and database types as well as to assign classes to databases. The metadata generator can operate in *incremental mode* in which case any changes made to the metadata by the programmer will be retained as metadata is generated for new class members and associations.

Persistence Context

All persistence activity occurs within a *persistence context*. The methods used by a programmer to find, create, and delete persistent objects as well as commit or rollback transactions are provided through the PersistenceContext class. Each persistence context contains a private object cache; therefore, the persistence activity of one thread can easily be isolated from the activity of other threads.

Persistence Manager

All persistence activity is coordinated by the *persistence manager*. The persistence manager loads the metadata required to map objects to tables and columns in the database, manages pools of persistence contexts and database connections, and generates new object identifiers. The PersistenceManager class also provides public methods to initialize and shutdown the persistence framework and to obtain a persistence context.

Persistent Objects

XStore does not require all persistent classes be derived from a persistent object base class; however, it does require a mechanism known as a *dirty flag* for tracking which objects have been modified. The developer of a class may choose to manage the dirty flag directly, in which case the class must either implement the *PObject* interface or extend the AbstractPObject class. In either case, the class should set the dirty flag whenever any persistent property of the object is modified. The developer may also choose to have the dirty flag managed by the proxy class. This approach may result in unnecessary database activity since the proxy class must assume that any method call which might modify the object actually did modify the object. In some applications the developer can achieve better performance by directly managing the dirty flag and only setting it if the object was indeed modified.

Transient Objects

In some applications it is desirable to create a transient object and later determine if the object should be persisted. By default, the factory and proxy classes generated by XStore create persistent objects that immediately become part of the current transaction; however, the factory and proxy classes also support the creation of transient objects. A transient object is accessed through a proxy object and resides in the object cache like a persistent object, but is marked transient and does not become part of the current transaction unless the application later requests that the object be persisted. Unlike persistent objects, transient objects remain in the cache after a commit or rollback. The most convenient way to create a transient object is to use the createTransient method of the factory class generated by XStore.

Persistence Domains

A *persistence domain* is one or more databases managed by one or more instances of the persistence manager. An object can be created in any database within a persistence domain provided the database contains the tables necessary to persist the object. Based on the metadata provided, XStore can determine which databases exist within a domain, which tables exist in each database, and which tables are required to persist a class. An exception is thrown if an attempt is made to store an object in a database that does not contain the necessary tables to persist it.

An object can have associations to objects in other databases. The *persistence domain administrator* may choose to spread classes across multiple databases to improve performance or simplify maintenance. For example, Employee objects could be assigned to database A and have associations to Address objects assigned to database B. XStore always knows where to find an object and how to load it because the LOID contains the database ID and class ID.

Sequencers

Each persistence domain must contain one database which stores the mapping metadata and sequencers used by XStore. The sequencers are used to generate unique identifiers for each class, database, and object. Each instance of the persistence manager maintains a connection to this metadata database which is initially used to load the metadata and later to access the OID sequencer when creating new persistent objects. The class and database sequencers are used only by the metadata generator and editor tools.

To improve performance, OID generation is a distributed task. Each persistence manager accesses the OID sequencer and reserves a block of OIDs. The persistence manager assigns OIDs from this block until the block is exhausted thereby reducing traffic between the persistence manager and the sequencer. The block size or *stride* is configurable. A distributed OID sequencer also improves reliability since any instance of the persistence manager can be shut down without affecting other instances. A fault-tolerant database can be used to host the OID sequencer further eliminating single points of failure.

Finding Objects

The PersistenceContext class provides a find method that locates persistent objects matching some search criteria and returns a collection of proxies for the matching objects. The Criteria class encapsulates the search criteria. The following example returns a collection of each Employee with the surname “Jetson”.

```
Criteria crit = pm.createCriteria(“com.spacely.Employee”);
crit.equal("surname","Jetson");
Collection jetsons = pm.find(crit);
```

We can also search for objects that match patterns with wild cards. This example will find all employees with a surname that begins with the letter ‘J’.

```
Criteria crit = pm.createCriteria(“com.spacely.Employee”);
crit.like("surname","J%");
Collection j = pm.find(crit);
```

Criteria can be arbitrarily complex. Here is an example that finds all male employees who are between the ages of 30 and 40 with a first name starting with the letter “T”.

```
Criteria cr1 = pm.createCriteria(“com.spacely.Employee”);
cr1.equal(“sex”, “M”).between(“age”,30,40).like(“firstName”, “T%”);
Collection employees = pm.find(cr1);
```

The programmer specifies search criteria in terms of classes, attributes. and associations rather than tables and columns. When generating the query, XStore maps names from the

object domain to the relational database domain. The criteria defined in the preceding example might generate the following query.

```
select FIRST_NAME, LAST_NAME, AGE, SEX, ...
from EMPLOYEE
where SEX = 'M' and
      AGE between 30 and 40 and
      FIRST_NAME like 'T%';
```

Deleting Objects

Since persistent objects continue to exist in the database after a program terminates, they must be explicitly deleted when no longer needed. PersistenceContext provides methods to delete persistent objects given a LOID, a proxy, or a collection of proxies.

CHAPTER 2

Class Preparation

Interface Extraction

Before we can persist an object using XStore, we must extract an interface from the class definition. Programmers with EJB or CORBA experience will find this a familiar process. For any class X, we want to rename it to a name such as XImpl (for 'X Implementation'), then create an interface X that declares every method implemented by the original class X. Some development environments provide an interface extraction tool; however, such tools may not retain the informative Javadoc comments every programmer dutifully includes for the benefit of others. It is preferable that a user not be required to study the implementation to understand the interface. For this reason, you might prefer simply editing a copy of the original class and removing everything between and including the opening and closing brace of each method, leaving the method signature along with the Javadoc comments.

Modification Tracking

XStore requires that every implementation object provide a mechanism known as a *dirty flag* for tracking which objects have been modified. This can be provided by implementing the *PObject* interface, extending the AbstractPObject class, or by creating a proxy that manages the dirty flag. The *PObject* interface declares the methods isDirty, clearDirty, setDirty, and post$load. AbstractPObject implements these methods and provides the boolean _dirty flag. Whether the user implements *PObject* or extends AbstractPObject, it is necessary to call the setDirty method in each method that modifies the object.

In the following example, we begin with a simple Person class that is to be persisted.

```
public class Person {
  /*
   * Construct a Person object.
   */
  public Person(String firstName, String surname) {
    this.firstName = firstName;
    this.surname = surname;
  }

  /*
   * Copy the properties of a Person to this object.
   * @param  person  the Person to be copied.
   */
  public void copy(Person person) {
    this.firstName = person.firstName;
    this.surname = person.surname;
  }

  /*
   * Returns the person's first name.
   * @return the person's first name.
   */
  public String getFirstName() {
    return this.firstName;
  }

  /*
   * Sets the person's first name.
   * @param name  the person's new first name.
   */
  public void setFirstName(String name) {
    this.firstName = name;
  }

  /*
   * Returns the person's surname.
   * @return the person's surname.
   */
  public String getSurname() {
    return this.surname;
  }
```

```
  /*
   * Sets the person's surname.
   * @param name  the person's new surname.
   */
  public void setSurname(String name) {
    this.surname = name;
  }

  private String firstName;
  private String surname;
}
```

From this class we extract the Person interface.

```
public interface Person {
  /*
   * Copy the properties of a Person to this object.
   * @param  person  the Person to be copied.
   */
  public void copy(Person person);

  /*
   * Returns the person's first name.
   * @return the person's first name.
   */
  public String getFirstName();

  /*
   * Sets the person's first name.
   * @param name  the person's new first name.
   */
  public void setFirstName(String name);

  /*
   * Returns the person's surname.
   * @return the person's surname.
   */
  public String getSurname();

  /*
   * Sets the person's surname.
   * @param name  the person's new surname.
   */
  public void setSurname(String name);
}
```

The original Person class is renamed PersonImpl. The modified class implements the Person interface we just created and extends AbstractPObject. The setFirstName, setSurname and copy methods were modified to call setDirty. Note the implementation of copy. Since a Person is not necessarily a PersonImpl, and is most likely a proxy, we cannot assume that we can directly access the firstName and surname properties.

```
/**
 * The new, improved, *persistable* PersonImpl class!
 */
public class PersonImpl extends AbstractPObject implements Person {
  /*
   * Construct a PersonImpl object.
   */
  public PersonImpl(String firstName, String surname) {
    this.firstName = firstName;
    this.surname = surname;
  }

  /*
   * Copy the properties of a Person to this object.
   * @param  person  the Person object to be copied.
   */
  public void copy(Person person) {
    this.firstName = person.getFirstName();
    this.surname = person.getSurname();
    setDirty();
  }

  /*
   * Returns the person's first name.
   * @return the person's first name.
   */
  public String getFirstName() {
    return this.firstName;
  }

  /*
   * Sets the person's first name.
   * @param name  the person's new first name.
   */
  public void setFirstName(String name) {
    this.firstName = name;
    setDirty();
  }
```

```
  /*
   * Returns the person's surname.
   * @return the person's surname.
   */
  public String getSurname() {
    return this.surname;
  }

  /*
   * Sets the person's surname.
   * @param name  the person's new surname.
   */
  public void setSurname(String name) {
    this.surname = name;
    setDirty();
  }

  private String firstName;
  private String surname;
}
```

Improving Performance

To improve performance, the PersonImpl class could be more judicious about setting the dirty flag. For example, suppose setFirstName is called with the string “John”. If the value of the firstName property is already “John”, then there is no need to set the dirty flag causing the object to be saved to the database. The setFirstName method could be rewritten as follows.

```
  /*
   * Sets the person's first name.
   * @param name  the person's new first name.
   */
  public void setFirstName(String name) {
    if ( (firstName != null && !firstName.equals(name))
        || (firstName == null && name != null) ) {
      firstName = name;
      setDirty();
    }
  }
```

Implementing PObject

In this example we were able to extend AbstractPObject because Person did not previously have a base class. Since Java does not support multiple inheritance, any class that already

extends a base class will not be able to extend AbstractPObject and must implement *PObject*. This can be done by including the following code.

```
public boolean isDirty() {  return dirty; }
public void clearDirty() {  dirty = false; }
public void setDirty() {  dirty = true; }
public void post$load() { }
public transient boolean dirty;
```

Common Mistakes

The most common mistakes made when creating persistent classes are failing to set the dirty flag and assuming an object passed as a parameter is an implementation object. If changes to an object do not appear to be saved to the database after calling PerrsistentContext.commit, the first thing a programmer should check is if all methods which modify the object set the dirty flag. The second mistake is often much more difficult to find. If a parameter is a persistent type, then the object to which it refers will almost certainly be a proxy and not an implementation object. The safe approach is to always write methods in terms of the interface and never assume you can directly access the properties of an object.

CHAPTER 3

Generating Proxy and Factory Classes

Using the Proxy Generator

Persistent objects are only accessed through a proxy; therefore, a proxy class is required for each persistent class. XStore provides a tool for generating proxy classes called ProxyGenerator. This tool analyzes the implementation class and generates a proxy class with constructors and methods that match the methods of the implementation class. Each proxy method checks the object cache directory and attempts to load the object if it is not already present in the cache. Once the object is loaded, the proxy method delegates to the implementation method. Any value returned by the implementation method is returned by the proxy method. A proxy constructor creates the implementation object, passing its parameters to the implementation constructor. It also registers the object with the persistence manager which assigns it an OID and a slot in the object cache directory.

ProxyGenerator can also generate a factory class for each persistent class. For each implementation constructor, the factory class will have create and createTransient methods with the same parameters. While XStore does not require the use of factories, it is the most convenient way to create a transient object. The createTransient method returns a proxy for a transient object that resides in the object cache but is not persistent. The application can later make the object persistent by calling the `persist` method of PersistenceContext and passing the proxy for the transient object as a parameter.

ProxyGenerator is used as follows.

```
PB=com.xavax.xstore.tools.ProxyGenerator
java $PB package interface implementation proxy [factory]
```

For example, the following command would create the proxy and factory classes for Spacely Sprocket Corporation's Address business object.

```
java $PB com.yoyodyne Address AddressImpl AddressProxy AddressFactory
```

Automating Proxy Generation Using Ant

It is usually preferable to generate proxy and factory classes as part of the automated build process for a project. Here is are example targets from an Ant build.xml file that creates the proxy and factory classes for Yoyodyne's business objects.

```
<property  name="ProxyGenerator"
           value="com.xavax.xstore.tools.ProxyGenerator"/>

<target name="proxyBuild" description="Build business object proxy">
  <java classname="${ProxyGenerator}">
    <classpath refid="proxy.path" />
    <arg value="${package}"/>
    <arg value="${stem}"/>
    <arg value="${stem}Impl"/>
    <arg value="${stem}Proxy"/>
    <arg value="${stem}Factory"/>
  </java>
</target>

<target name="proxies" depends="jar,demojar"
        description="Build and compile business object proxies">
  <antcall target="proxyBuild">
    <param name="package" value="com.spacely"/>
    <param name="stem" value="Address"/>
  </antcall>
  <antcall target="proxyBuild">
    <param name="package" value="com.spacely"/>
    <param name="stem" value="Employee"/>
   </antcall>
  <antcall target="proxyBuild">
    <param name="package" value="com.spacely"/>
    <param name="stem" value="Sprocket"/>
   </antcall>
</target>
```

In this example, the `proxy.path` property would include all class directories and jar files that are needed to load the implementation classes for Address, Employee, and Spaceship. ProxyGenerator uses reflection to discover the constructors and methods of the implementation class so it is necessary that it be able to load the implementation class.

Proxy Implementation

The programmer seldom needs to be concerned with the details of the proxies generated by XStore/J; however, when stepping through a program using a debugger, it is useful to know the basic structure of a proxy. Here is a fragment of a proxy class generated for the Person class described in chapter 2.

```
final public class PersonProxy extends AbstractProxy implements Person {
  public PersonProxy(AccessKey key) {
    super(key);
  }
  ...
  public PersonProxy(String p0, String p1)
    throws PersistenceException
  {
    super(new PersonImpl(p0, p1));
  }

  public String getFirstName()
  {
    return ((Person) $link.access()).getFirstName();
  }

  public void setFirstName(String p0)
  {
    ((Person) $link.access()).setFirstName(p0);
  }
  ...
  public int hashCode()
  {
    return ((Person) $link.access()).hashCode();
  }
  ...
}
```

The heart of a proxy is the expression `$link.access()`. Each proxy inherits a protected member `$link` of type Link. A Link is an entry in the object cache directory and provides an `access` method which determines if a persistent object is in the object cache and if not, tries

to load it, then returns the object. Once the object is returned, the proxy proceeds to call a method of the object.

For each constructor in the implementation class, the proxy class has a constructor with the same parameters; therefore, the user can simply change Person to PersonProxy in code that creates Persons. A proxy includes one additional constructor which is only used by XStore. The need to limit access to this constructor poses a problem. The proxy does not belong to any XStore package; therefore, we cannot use package level access, and for the same reason the constructor has to have public access. We solve this problem by using the *private interface* design pattern.The constructor has a parameter of type AccessKey. This class is part of the main XStore package and only has constructors with package level access. Since no class outside of XStore can create an AccessKey, the proxy constructor can be public but only an XStore class can call it.

Note that the proxy implements the `hashCode` method. If this were not the case, it would inherit the implementation provided by `Object` which is almost certainly not the desired behavior. What the application really wants is the hash code of the implementation object and XStore facilitates this by having the proxy delegate the method to the implementation object. The method `toString` is implemented in a similar manner.

The implementation of the `equals` method provided by the AbstractProxy base class is significantly more complicated. Suppose we want to compare a proxy **p** of class P which refers to the implementation object $\mathbf{p}_{impl}$ with an object **o** which could possibly be a proxy for the implementation object $\mathbf{o}_{impl}$. The objects **p** and **o** are defined to be equal if and only if:

- **p** and **o** are the same proxy (**p** == **o**), or
- **o** is assignable to P, that is, **o** is also a proxy of the same class or a derived class as **p**, and $\mathbf{p}_{impl}$ and $\mathbf{o}_{impl}$ are the same object ($\mathbf{p}_{impl} == \mathbf{o}_{impl}$), or
- the previous condition is met and $\mathbf{p}_{impl}$.equals($\mathbf{o}_{impl}$) returns true, or
- $\mathbf{p}_{impl}$.equals(**o**) returns true.

CHAPTER 4

Generating Metadata

Using the Metadata Generator

Before an object can be persisted, XStore requires metadata that describes how members of the object are mapped to columns in the database. XStore provides a *metadata generator* tool which will analyze the classes to be persisted and generate both the mapping metadata as well as a suggested database schema. The output of the metadata generator consists of a file of SQL statements which can easily be edited by the programmer. Here is the command line syntax for running the metadata generator.

```
MB=com.xavax.xstore.tools.MetadataGenerator
java ${MB} [-incr] -input inputFilename -output outputFilename
```

The input file contains a list of class and data source specifications. The format of each data source specification is as follows.

```
database name {MSS|MySQL|Oracle} jndi-url
```

The format of each class specification is as follows.

```
class packageName interfaceName implementationName proxyName
```

The metadata generator uses reflection to determine the fields in each class; therefore, it is necessary for all classes or jar files needed to load the classes listed in the input file be included in the class path when the metadata generator executes.

Assumptions

The metadata generator makes the following assumptions.

- Each class maps to a table with the same name.
- Each attribute is mapped to a column in the table with the same capitalized name (e.g. age maps to Age).
- Each attribute of a primitive type (e.g. long) or a wrapper type (e.g. Long) is mapped to the most appropriate database-specific type.
- Each Date attribute is mapped to a database-specific type (e.g. DATETIME for MSSQL).
- Each String attribute is mapped to a VARCHAR(256).
- Each reference to an object of a persistent type is mapped to a set of three columns which will store the LOID of the object. The column names are created by concatenating the capitalized name of the member with the strings DB , Class , and OID (e.g. manager maps to ManagerDB, ManagerClass, and ManagerOID).
- Each reference to a collection of objects maps to a new table. The table name is chosen by concatenating the class name with the member name. For example, if a Person class has a reference to a collection named addresses, the association table would be called PersonAddresses. This table would have five columns. The first two are named srcOID and sequence. The remaining three names are generated as described above.
- If a class extends a class that is persistent, the fully normalized persistence strategy is used and a column named OID is added to the table.
- If inheritance is not used, three columns named databaseID, classID, and OID are added to the table.

Editing the Metadata

The output of the metadata generator is a file of SQL statements. The mappings determined by the generator can be modified by editing these statements. An interactive metadata editor tool is also under development.

Incremental Metadata Generation

If the -incr flag is included on the command line, the metadata generator runs in incremental mode. When running in incremental mode, the generator first loads the existing metadata. For each class listed in the input file, it determines if the class has existing metadata. If existing metadata is found, it only generates new metadata for any attributes or associations added to the class. If a programmer has previously edited the metadata, those changes are preserved.

CHAPTER 5

Programming with the XStore/J Framework

Initialization

To begin using XStore a program must first get a PersistenceManager. The most common way to do this is by calling the static method PersistenceManager.getManager and passing the name of a properties file located anywhere in your class path.

```
PersistenceManager pm = PersistenceManager.getManager("DemoApp");
```

In this example the file DemoApp.properties includes the properties necessary to initialize the XStore framework. Here is the minimum set of properties required.

java.naming.factory.initial
The factory used to create the initial naming context. If no name server is available, use com.xavax.xstore.util.InitialContextFactory.

java.naming.provider.url
The URL for JNDI name resolution. This is typically an LDAP server.

com.xavax.xstore.url
The URL for the data source used to get connections to the metadata database.

com.xavax.xstore.poolSize
The size of the pool of persistence contexts. A standalone application will need one persistence context per thread that uses persistence. When XStore is used in a container environment such as Tomcat or Weblogic, each concurrent thread will need a persistence context.

Here is a list of properties file that might be used in a typical Weblogic server environment.

```
java.naming.provider.url: t3://localhost:7001
java.naming.factory.initial: weblogic.jndi.WLInitialContextFactory
com.xavax.xstore.poolSize: 16
com.xavax.xstore.url: metadata
```

Obtaining a Persistence Context

Once we have a persistence manager, we can use it to obtain a persistence context by calling PersistenceManager.getContext.

```
PersistenceContext pctx = pm.getContext();
```

The persistence context returned by getContext is bound to the current thread, so any persistence activity in the current thread will automatically use this context.

Setting a Default Database

Before we can create objects, we need to specify the default database where the objects will be persisted. This is done using the PersistenceContext.setDefaultDatabase method.

```
pctx.setDefaultDatabase("SS_Test1");
```

Creating Objects

Persistent objects are created in one of two ways. When a proxy is created, an implementation object is automatically created and assigned a slot in the object cache directory. If we chose to create factories, we can also use one of the factory's create or createTransient methods. Here is how we might use the proxies and factories created in the chapter 3 to create a new Spacely Sprockets employee and assign his address.

```
Employee employee = new EmployeeProxy(“George”, “S”, “Jetson”);
Address address =
  AddressFactory.create(“1 Terrestrial Plaza”, “2201”, “Astoria”,
                        “Europa Colony”, “297028-2201”, “Jupiter”);
employee.setAddress(address);
employee.setAge(47);
employee.setSSN(“419-97-8894-3027”);
pctx.commit();
```

We now have an Employee object with an association to an Address persisted in the database SS_Test1. If we had called rollback instead of commit, the two objects would not be persisted and would have been removed from the object cache.

Locating Persistent Objects

Persistent objects are located using the PersistenceContext.find method or by association. Before we can use the find method, we must create a Criteria and initialize it with our search criteria. Here is how we would locate the Employee we just created using the find method and then locate his Address by association.

```
Criteria criteria = pctx.createCriteria("com.spacely.Employee");
criteria.equal("surname", "Jetson").equal("firstName", George);
Collection c = pctx.find(criteria);
if ( c.size() > 0 ) {
  employee = (Employee) c.get(0);
  address = employee.getAddress();
}
```

Assuming the find method returns a collection of at least one Employee, the variable employee now holds a proxy that the first Employee and address holds a proxy to their Address. When the program first tries to use address, that object will be loaded into the object cache.

Criteria can be arbitrarily complex. Most Criteria methods return the Criteria itself which is why we are able to chain the two calls to the equal method in our example. The two search criteria are logically ANDed. Criteria also supports the logical OR and NOT operations. Here is how we might create a Criteria to find all employees with a surname beginning with 'J' over 40 years old, or, employees with the first name "Elroy" under 40 years old.

```
criteria.reset();
criteria.like("surname", "J%").greaterEqual("age", 40)
        .or().equal("firstName", "Elroy").less("age" 40);
```

A Criteria is initially configured to search in the current default database. We can search for objects in another database by calling the Criteria.database method; however, we can only search in one database with each call to find.

Deleting Persistent Objects

Persistent objects continue to exist in the database after a program no longer refers to them; therefore, they must be deleted explicitly. PersistenceContext provides methods to delete an object given its LOID or a proxy for the object. Here is how we could change the address of our employee and delete the old address, assuming the variable employee still holds a proxy to the employee.

```
Address oldAddress = employee.getAddress();
employee.setAddress(newAddress);
pctx.delete(oldAddress);
pctx.commit();
```

Here is how we would delete all employees with the surname Jetson.

```
criteria.reset();
criteria.equal("surname", "Jetson");
Collection c = pctx.find(criteria);
pctx.delete(c);
```

Using Multiple Threads

XStore is thread safe. When multiple threads are used in a program, each thread can create its own context or one or more threads can share a context. A thread continues to share a context with its parent until it calls PersistenceManager.getContext to obtain a separate context. Any persistence activity performed by a thread in a context is isolated from any persistence activity occurring in other threads in other contexts.

Releasing Resources

A persistence context uses a lot of system resources; therefore, a program should release this resource when it is no longer needed. This is done by calling PersistenceContext.close. Before calling close the program should commit any changes; otherwise, the changes are lost as if the program called rollback. A standalone program should call PersistenceManager.shutdown to gracefully shut down the persistence framework.

CHAPTER 6

Understanding XStore Mapping Metadata

Overview

Metadata is data that describes other data. Metadata is used in XStore to describe how the data that makes up an object is structured and how that data should be mapped to tables in a relational database. XStore metadata is also stored in a database although the programmer can view and edit it in other forms. The purpose of this chapter is to describe the structure of the metadata and provide the programmer with an understanding of how to modify the metadata as necessary to accommodate the specific needs of the object model or map to a specific database schema.

XStore metadata consists of a hierarchy of mapping objects which represent each class, field, and association in the object model as well as each database, table, and column in the data model. The metadata object model is shown in Figure 5 on page 6-2.

PersistenceManager

The PersistenceManager loads the metadata from the database, resolves mapping references during the initialization of the framework, and provides methods for locating the metadata for a class, database, or table. The PersistenceManager also manages the sequencers used to generate unique identifiers for classes, databases, and tables.

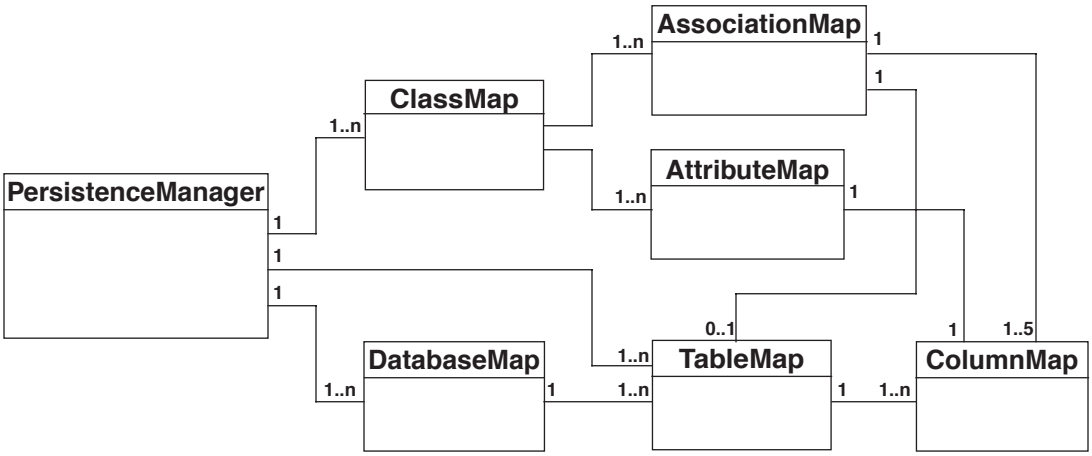

FIGURE 5. **XStore Metadata Object Model**

ClassMap

ClassMap describes the structure of a persistent class. The fields of a ClassMap, as stored in the table xClasses, are listed in Table 1. When loaded in memory, the names in the tableName, kdb, kclass, and koid fields are resolved to references to the appropriate TableMap and ColumnMap objects. The basename of the interface is used as the name of the class. After the metadata for associations and attributes is loaded and resolved, each ClassMap is populated with collections of AssociationMap and AttributeMap objects.

TABLE 1. **ClassMap Fields**

Name	Type	Description
classID	int(16)	The class identifier for this class.
parent	int(16)	The class identifier for the parent class
interface	string(128)	The fully-qualified interface name.
implementation	string(128)	The fully-qualified implementation class name.
proxy	string(128)	The fully-qualified proxy class name.
tableName	string(64)	The name of the table to which this class is mapped.
kdb	string(64)	The name of the column holding the database ID.
kclass	string(64)	The name of the column holding the class ID.
koid	string(64)	The name of the column holding the object ID.

AssociationMap

AssociationMap describes the structure and mapping of an association between one or more persistent objects. The fields of an AssociationMap, as stored in the table xAssociations, are shown in Table 2. When loaded in memory, the tableName, sourceOID, sequence, destDB, destClass, and destOID names are resolved to the appropriate TableMap and ColumnMap objects. The tableName, sourceOID and sequence fields are only used for 1:n associations.

TABLE 2. AssociationMap Fields

Name	Type	Description
classID	int(16)	The class identifier.
assocID	int(16)	The association identifier.
name	string(128)	The name of the association.
dependant	boolean	True if the associated object is dependant on the source object.
multiplicity	boolean	True if this is a 1:n association.
tableName	string(128)	The name of the table to which this association is mapped.
sourceOID	string(64)	The name of the column holding the OID of the source object.
sequence	string(64)	The name of the column holding the sequence number.
destDB	string(64)	The name of the column holding the database ID of the destination object.
destClass	string(64)	The name of the column holding the class ID of the destination object.
destOID	string(64)	The name of the column holding the object ID of the destination object.

AttributeMap

AttributeMap describes the mapping of an attribute. The fields of an AttributeMap, as stored in the table xAttributes, are shown in Table 3. When loaded into memory, columnName is resolved to a ColumnMap object.

TABLE 3. AttributeMap Fields

Name	Type	Description
classID	int(16)	The class identifier.
attrID	int(16)	The attribute identifier.
name	string(64)	The name of the attribute.
columnName	string(64)	The name of the column holding the attribute data.

TableMap

TableMap describes a table available in one or more databases in a persistence domain. The fields of a TableMap, as stored in the table xTables, are shown in Table 4. After the metadata for columns is loaded, each TableMap is populated with a collection of ColumnMap objects.

TABLE 4. TableMap Fields

Name	Type	Description
tableID	int(16)	The table identifier.
name	string(128)	The table name.

ColumnMap

ColumnMap describes a column in a table. The fields of a ColumnMap, as stored in the xColumns table, are shown in Table 5.

TABLE 5. ColumnMap Fields

Name	Type	Description
tableID	int(16)	The table identifier.
columnID	int(16)	The column identifier.
name	string(64)	The name of the column.
type	int(8)	The SQL type of the column (see Table 9 on page 6-5).
width	int(32)	The width of the column.
precision	int(8)	The precision of the column.

DatabaseMap

DatabaseMap describes a database in a persistence domain. The fields of a DatabaseMap, as stored in the xDatabases table, are shown in Table 6.

TABLE 6. DatabaseMap Fields

Name	Type	Description
dbID	int(16)	The database identifier.
name	string(32)	The database name.
url	string(255)	The URL for finding the data source.
style	string(16)	The locking style to use with this database.

DatabaseTables

DatabaseTables is a cross-reference table used to map tables to databases. The fields of DatabaseTables, as stored in the table xDatabaseTables, are shown in Table 7.

TABLE 7. DatabaseTables Fields

Name	Type	Description
dbID	int(16)	The database identifier.
tableID	int(16)	The table identifier.

Sequences

Sequences are objects used to generate unique identifiers. The fields of a sequence, as stored in the xSequences table, are shown in Table 8.

TABLE 8. Sequence Fields

Name	Type	Description
seqID	int(16)	The sequence identifier.
name	string(32)	The sequence name.
sequence	int(64)	The next available sequence value.
stride	int(16)	The number of identifiers to allocate per request.

SQLTypes

SQLTypes is a cross-reference table mapping SQL type numbers used internally by XStore to SQL type names. The list of SQL type numbers is shown in Table 9.

TABLE 9. SQLTypes

Type	Name	Type	Name	Type	Name
1	ARRAY	8	CLOB	15	NULL
2	BIGTINT	9	DATE	17	SMALLINT
3	BINARY	10	DOUBLE	18	TIME
4	BIT	11	FLOAT	19	TIMESTAMP
5	BLOB	12	INTEGER	20	TINYINT
6	BOOLEAN	13	LONGVARBINARY	21	VARBINARY
7	CHAR	14	LONGVARCHAR	22	VARCHAR

Editing Metadata

The metadata generator produces metadata in the form of SQL statements which can be edited by the programmer. The most common reason to edit the metadata is to adapt to a different database schema. This is easily accommodated by editing the database, table, and column names appearing in the SQL statements. Errors in the metadata are difficult to diagnose and place data integrity at risk; therefore, the following precautions should be observed.

- Avoid making changes to metadata after a persistence environment begins operation.
- Avoid making changes to metadata of a persistence environment when one or more applications which use that environment are operating. This could result in applications using different metadata.
- Avoid changing numeric identifiers appearing in the metadata.
- Never change the value of a sequencer (the sequence column) after a system begins operation. Changing the sequencer value could result in two or more classes or objects having the same identifier.

Running the metadata generator in incremental mode after editing the metadata will catch some editing mistakes. See “Incremental Metadata Generation” on page 4-2 for more information about incremental mode.

CHAPTER 99

Planned Enhancements

Cache Performance

XStore currently uses a pessimistic caching strategy that assumes other applications in the environment can concurrently modify the database. Currently the object cache is part of the persistence context, so one context may load an object that is already in the cache of another context. After a commit or rollback, all objects are removed from the cache. A future version of XStore will implement a two-level cache strategy by adding a level 2 (L2) shared cache to the persistence manager while the persistence context will implement a level 1 (L1) non-shared *write-back* cache. When attempting to load an object, the persistence context will first check this L2 cache. If a hit occurs, it will simply refer to the object in the L2 cache. If a miss occurs, the object will be loaded into the L2 cache and the L1 cache. During a commit, any modified objects in the L1 cache are written back to the L2 cache and also persisted in the database.

Smart Containers

XStore currently supports using any container that implements the *Collection* interface to store associations. If an object is marked as modified in the object cache directory, the associations might have been modified so XStore deletes the old collection of associations from the database and saves the current collection. A smart container could improve performance by tracking exactly which elements of the collection were added or deleted and performing only the minimum set of database operations needed to update the collection.

Smart containers could also allow lazy loading of an association. Currently, XStore loads an object and all of its associations (but not the associated objects) on demand. This requires accessing one or more tables to load the object and one table for each 1:n association. When using a smart container, the OID of the parent object could be saved by the container and the contents would only be loaded when the program actually tried to use the container.

Association Table Reuse

XStore currently requires one table for each 1:n association. This can result in many similar tables. For example, a program managing a car rental company might need to associate each Location with a list of cars available, cars on lease, and cars undergoing maintenance. There are three 1:n associations so XStore uses three tables; however, each is an association from a Location to a Car. The association table currently includes the source LOID, destination LOID, and a sequence number. By adding an additional field which indicates to which association a row belongs, all three associations could be persisted in the same table.

Automatic Optimistic Locking

Keeping database rows or tables locked during long transactions can severely impact the performance of some applications. One solution to this problem is *optimistic locking*. When using optimistic locking, the rows or tables involved in a transaction are not actually locked but the version number or timestamp of each object is saved when the transaction begins. When the program attempts to update the object during a commit, it compares the current version with the saved version and throws an optimistic locking exception if the two versions are not the same. A future version of XStore will automate this process by adding an additional version number field to the tables used to persist each class that is configured to use optimistic locking. This version number will be saved in the object cache directory when the object is loaded. When the transaction is committed, XStore will generate an update statement that both checks and increments the version number.

Object Migration

As databases grow and requirements change, it may be necessary to rearrange the mapping of classes to databases. For example, as a business grows the number of Customer objects grow with it and eventually it may be necessary to assign an additional database server to store Customer objects. An alternative scenario might be consolidating objects from two existing servers onto a single, higher performance server.

While it is a simple matter to find a set of Customer objects and copy them to another database, we are left with two problems: identity preservation and dangling associations. A user program could simply delete a Customer from one database and create it in another; however,

that would result in a Customer with a different OID. The OID of an object should be immutable; therefore, it is necessary that the persistence framework provide a mechanism to migrate an object while preserving the OID.

After the object has migrated to its new home, any associations to this object are now broken or dangling since they still use a LOID containing the old database ID. Examining every object in an persistence domain that could possibly have an association to the objects being migrated is a formidable task that is further complicated by the fact that XStore supports inheritance and polymorphism. This task might occupy a significant portion of server resources possibly requiring any applications using the persistence domain to be shut down.

A future version of the XStore object loader will exploit the LOID structure to implement */azy migration*. When an object migrates to another database, a *forwarding object* will be left in the old location with a LOID containing the new database ID. When the object is loaded, the loader will recognize that the database ID in the LOID does not match the ID of the database from whence it was loaded. The object will then be loaded from its new location and the source of the association will be marked as dirty. When the transaction is committed, the source object will be saved with an association to the new LOID of the migrated object. At some point all broken associations will be repaired; however, to assure this, a background scrubber task can search for and broken associations. XStore's implementation of lazy migration will make it possible for a persistence domain to operate "24x7" even during major database reorganization.

Schema Evolution

As an application ages, the classes that make up the application often evolve by adding or deleting fields and associations; therefore, the database schema must evolve with it. This is problematic in a "24x7" environment as there may never be a maintenance window long enough to allow all objects to be migrated to the new schema. A future version of XStore will solve this problem by implementing *lazy schema evolution.* The existing LOID and metadata structures will be extended adding a schema version number (SVN). When an object is loaded into cache, XStore will compare the object's SVN with the latest SVN for the class. If the object uses an old schema, it will be marked dirty and XStore will attempt to evolve the object. In most cases this can be accomplished by setting new fields to default values and deleting old fields. In cases where a class changes radically or there is no default value, it may be necessary for the class to intervene. This will be accomplished using the post$load method which is already included in the PObject interface. When the transaction is committed, the object will be saved to the database with the new SVN. As with lazy object migration, a scrubber task can use idle system time to iterate over each instance of a class.

Annotations

XStore currently works with JDK 1.4 or 1.5; however, it does not use any features introduced in JDK 1.5. One of the more interesting new features is annotations. An annotation is metadata about a class, method, or field that is included in the class file when a class is compiled. The new Java Persistence API defines a set of annotations for encoding metadata about how an object should be persisted. A future version of XStore will use these standard annotations to affect the output of the metadata generator. For example, the metadata generator currently defaults to mapping a field to a column of the same name. With annotations, the programmer can provide the metadata generator with an alternative column name. It is important to note that the persistence domain administrator can still edit the metadata and override this name. This allows a program using XStore to adapt to different database schemas without having to ship the source code of the program.

EJB 3.0

The latest specification for Enterprise Java Beans (EJBs) supports the use of third party persistence solutions which integrate with the EJB container. A future version of XStore will provide the necessary interfaces to be used as an EJB 3.0 persistence solution.