

Persistent Java

C. Souza dos Santos* E. Theroude
INRIA O₂ Technology
(Cassio.Souza@inria.fr) (Eric.Theroude@o2tech.fr)

September 1996

Abstract

We report on the design and implementation of Persistent Java (hereafter PJ), a combination of the Java programming language with a relational database which brings persistence to Java objects in a transparent way. We give an overview of the main functionalities of the PJ API. We describe the ongoing implementation of PJ on top of a JDBC compliant database interface. This includes the automatic mapping of Java classes to a relational database format and the transparent data transfer between a relational data store and the Java application heap. We point to future directions.

1 Introduction

Persistent Java is a Licensed Technology implementing a Java-to-relational database binding that allows Java objects to be stored in relational databases. Although implemented by an underlying relational database system, the data store is perceived as a transparent persistent object repository by a PJ application. Deploying such an application includes three distinct main activities:

1. Setting up the underlying relational database, called a PJ database, where Java objects and class information are to be stored.

A set of tools is provided to allow the creation and initialization of a PJ database.

2. Import Java classes to the database

The import program `javai` is a tool allowing Java classes to become persistent capable. It creates and installs all database structures (tables, stored procedures) necessary to store instances of a given class or set of classes in a PJ database. It also patches imported classes by adding some methods to their interface. These methods are used by the application programs and by the PJ runtime system as well.

*INRIA industrial post-doc position at O₂ Technology.

3. Code applications using the PJ API and the methods generated by the import program.

A simple and yet powerful API is provided as a package of Java classes to allow the development of PJ applications. This API implements a database model where databases can be opened and closed, transactions can be started, committed and aborted, data (including metadata, i.e. bytecodes) can be written to or read from the database, users and permissions can be defined and the database catalog can be inspected.

Persistence is orthogonal to object types. Every Java class can be imported to a relational schema and every variable of a Java class can be imported regardless of its visibility. Variables declared as transient in a *configuration file* are not imported. Every Java object can thus be partially or entirely stored in a relational database. As static variables contents can also be stored in the database, they can be used to define roots of persistence (data entry points) in the database in the style of the ODMG model [2].

Figure 1 illustrates the general architecture of PJ.

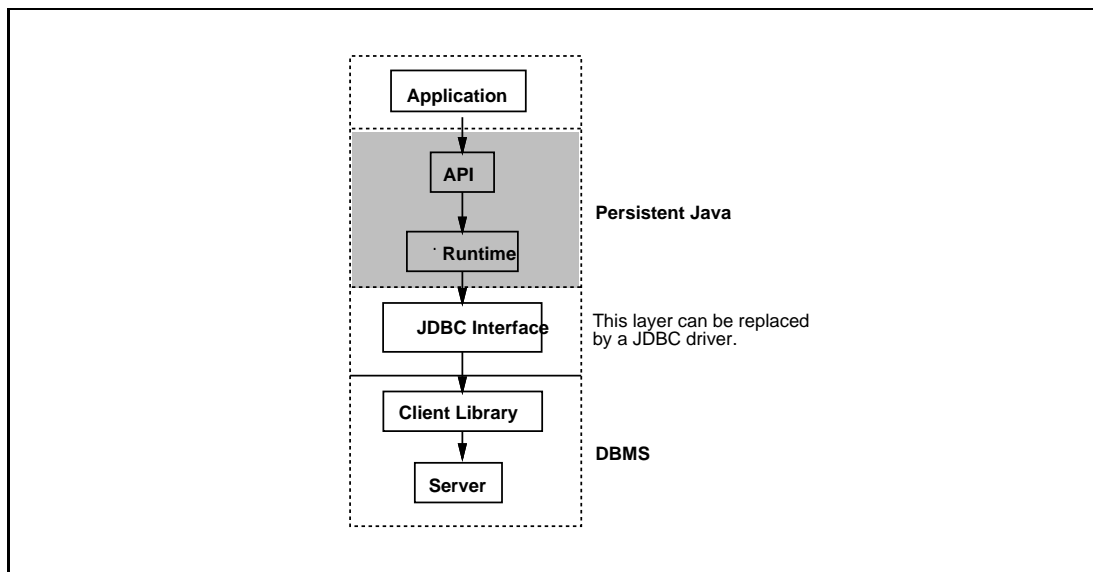


Figure 1: Architecture of Persistent Java

At the top level, user applications are written with standard Java using the PJ API. The API relies on the PJ runtime system, which is composed of a database cache and is in charge of controlling data transfers between the Java heap and the database. The PJ runtime uses, in turn, a JDBC interface [4] to communicate with the underlying relational database server.

This article is organized as follows. Section 2 describes the import program functionalities and overviews the mapping of Java classes into relational database structures. Section 3 presents the main components and features of the PJ API. Section 4 overviews the implementation of the runtime system and Section 5 points to future directions.

2 Import

Classes are imported through the `javai` import program. The import program generates and installs all relational database structures necessary to store/retrieve objects of a Java class into/from a database in an optimized and transparent way.

The usage of the `javai` import program, which is itself a Persistent Java application, is shown below. The meaning of its arguments is explained in the sequel.

```
javai [valid options and arguments]
  -system <$val> (set system type to $val (e.g. Oracle,Sybase)
  -server <$val> (set database server to $val)
  -base <$val> (set PJ database name to $val)
  -user <$val> (set database user to $val)
  -passwd <$val> (set user password to $val)
  -config <$val> (set configuration file to $val)
  -boot (enable boot of PJ base)
  -input <$val> (set path where classes to be imported are to be searched)
  -meta (enable storage of bytecodes)
  -delete (enable deletion of classes or bytecode from PJ base)
  -dumpclass (inspect the structure of imported classes)
  -dumpsqll (inspect the relational structure of imported classes)
  -output <$val> (set the output directory for dump options to $val)
  -package <$val> (a package name, whose classes are to be imported or deleted)
  <$val> (a list of qualified class names to be imported or deleted)
```

The import program generates some Java source code for each imported class. This code corresponds to the implementation of methods declared in the interface `PersistentObject`, which is defined in the PJ API package, and that every imported class must implement.

A Persistent Java application does not need to know about database structures that are generated when a class is imported. These data structures are used by the runtime to perform data transfer between an application and the database. The application program relies on a set of high level primitives to store and retrieve Java objects.

By separating the import phase from the database interface runtime, we were able to considerably improve the performance of the latter. At runtime, all database structures allowing Java objects to become persistent must have been previously installed by the `javai` program.

2.1 Creating a database

An auxiliary tool called `create_importer` allows the creation of PJ database user, called a PJ *importer*, by a database system administrator. An importer can, in turn, create new PJ databases through the tool `create_base`. When a given importer creates a database, he or she becomes the *owner* of the database. Only the database owner has the permission to import classes into a given database.

2.2 Booting a database

Once a new database is created, it must be initialized so that classes can be imported into it. The database initialization is performed by the `javai` program with option `-boot`. When this option is set, no class is imported and the database is simply initialized. An error is reported if the database has been already initialized. Again, only the database owner is allowed to initialize a database.

Database initialization actually includes the import of some classes, the so-called catalog classes, but these are hidden from the user. The initialization consists of installing the system catalog, i.e. the database structures that are used to store information about imported classes. This catalog is implemented by Java objects that are stored in the database and, in that sense, the `javai` program is itself a PJ application, as it uses the PJ API to retrieve and update catalog information.

2.3 The configuration file

A configuration file allows the user to interfere with the default import mechanism. This way, the user can filter class variables that are to be stored in the database, by declaring some variables as *transient*, and/or provide some information that allow the database system to optimize the way variables of type `String` and `byte[]` are stored.

The variables of a given object are stored based on a default type mapping, that gives for each Java type a corresponding database type. The user can interfere with the default type mapping to allow the storage of arbitrarily long text and image data in an optimized way.

The configuration file can also be used to define some renamings. This allows default naming rules to be redefined by the user, so as to avoid naming conflicts with particular database reserved words or to allow database structures, i.e. tables, to be generated according to some specific user's preferences. This is in general not necessary, as such structures are hidden from the user relying on the API to access and store objects in the database, but knowing the relational database structure can be useful in some situations where the database should be accessed directly rather than through the API.

2.4 Inspecting the catalog

The import program can be used to inspect the system catalog. The option `-dumpsq1` allows the relational tables generated for the imported classes to be inspected in a given database. The option `-dumpclass` allows the imported classes themselves to be inspected. The user can thus check which classes are imported in a given database and *how* they are imported, i.e., if renaming and hiding of variables, for instance, were defined when the classes were imported.

2.5 Unimporting

Imported classes can also be deleted from the database by setting the option `-delete` of `javai`. This is allowed only if there are no instances of the class to be deleted nor

of one of its subclasses, and if no other imported class (not being itself deleted) points to it through one of its persistent variables¹.

2.6 Reimporting

Imported classes can evolve and be reimported so that the database take into account their evolution. Limited support for class evolution is provided for the moment, namely the addition of new variables. New variables can be added to an imported class without loss of existing data. For instances of the class (or of one of its subclasses) already stored in the database, such variables are initialized with the corresponding default values.

2.7 Mapping of Java classes

The mapping of a set of classes into a relational schema has been considered in a number of works, e.g. [7, 6]. Our mapping not very different from existing proposals but presents some particularities that are related to specificities of the source OO model, i.e. the Java language, and of our runtime system.

The `java` import program performs the following steps when processing a Java class named `C`:

1. Generation of a *class table* named `C` into which local variables of `C` are mapped as columns following a default type mapping.
2. Generation of a *static table* named `C_S` where static variables, if any, of `C` are stored.
3. For each array variable `v` of type `T[]`, if any, generation of an *array table* `T_A` and a corresponding *array element table* `T_A.E` for storing arrays and array elements respectively.
4. Generation of stored procedures to perform insertion, deletion, selection and update of objects stored in the database. These procedures are called by the PJ runtime through the JDBC interface.
5. Generation of API methods, e.g. `access`, `update`, `delete`, `destroy`, in class `C`.

The mapping mechanism applies special rules for mapping inheritance, nested arrays and references between objects. Each class in an inheritance path is mapped to a table containing columns for local variables defined in the corresponding class only. The retrieval of an object involves the join of all the related rows in tables generated from its class and superclasses up to `Object`. Primary keys and indexes are used to speed up object retrieval.

Array variables are mapped into two auxiliary tables, one for storing the array identifier (which is system generated, as for ordinary object identifiers) and array size,

¹A variable of an imported class is said to be persistent if it is not declared as transient in the configuration file.

and another for storing the array elements. The storage and retrieval of array variables is optimized to avoid unnecessary loading of array elements stored in the database. In other words, array elements are loaded on demand.

References between objects are mapped to foreign keys and referential integrity is enforced by the underlying database so that an object pointed to by a persistent object cannot be deleted from the database. This way, we avoid dangling references in the database.

The generation of stored procedures (collections of SQL statements and control-of-flow language) at import time has multiple advantages. First, as an execution plan is prepared the first time the procedure is run, or, in other words, as stored procedures are precompiled, subsequent execution is optimized. Also, by relying on procedures, we reduce the amount of information to be passed from the Java runtime to the database server to a minimum. As it will become clearer in the sequel, the communication between the runtime system and the underlying database server is limited to calling system-generated stored procedures with appropriate parameters and retrieving returned results.

Example 2.1 The following shows how `javai` can be used to import class `Student`, given below, into a PJ database named `java_base` under the control of the user `java_importer`.

```
package example;
public class Person {
    int age;
    String name;
    Person spouse;
    Person children[];
}
public class Student extends Person {
    Person responsible;
}

javai -system Sybase -server java_store -base java_base
      -user java_importer -passwd java_admin example.Student
```

□

Figure 2 illustrates the different steps involved in the import of class `Student`. Note that class `Person`, the superclass of `Student`, is automatically imported into the database when `Student` is imported.

Numbers between parentheses indicate the order in which different tasks are performed. At the beginning, step 0, the user disposes of one or more Java classes whose instances he/she wants to make persistent. At step 1 he imports such class(es) to the database with the `javai` program. This program installs all database structures to allow the storage and retrieval of instances of the imported class(es) at step 2 and generates the implementation of methods to be called from applications in the corresponding

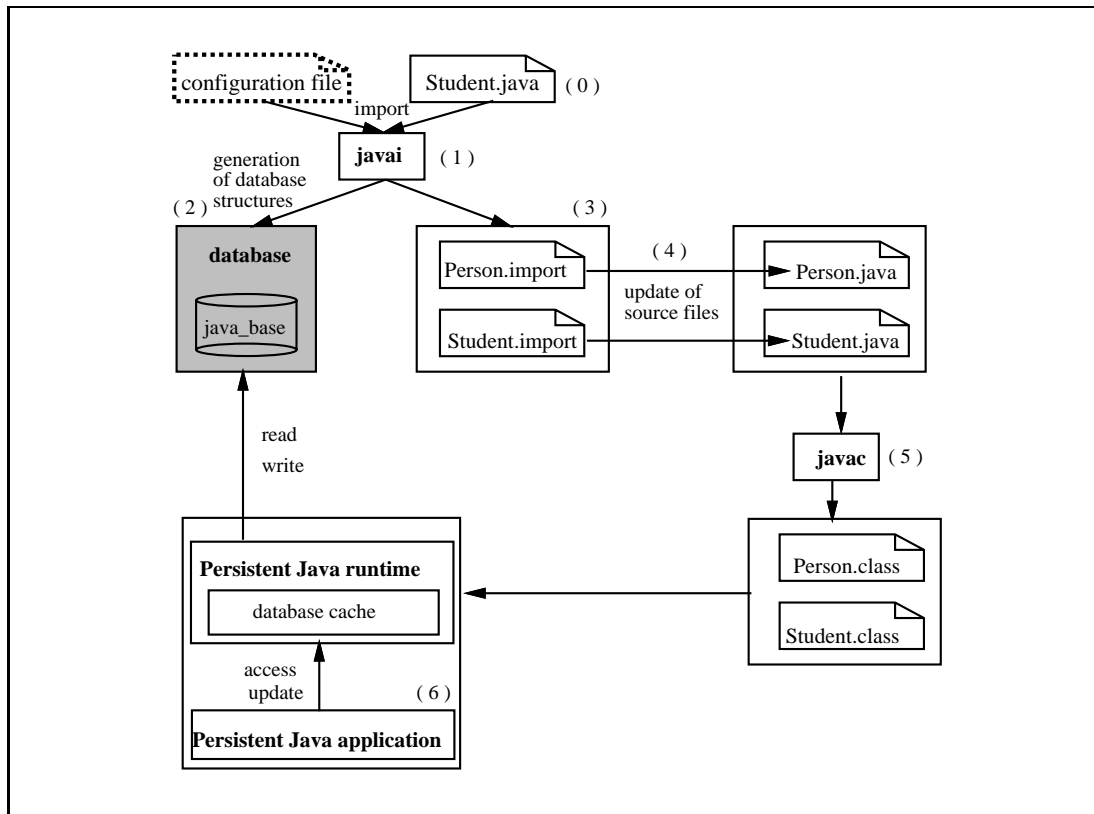


Figure 2: Importing of class Student

.import files. At step 4, the user places the generated code in the corresponding .java files and at step 5 these files are recompiled. At step 6 the user can compile and run a Persistent Java application using the PJ API described in Section 3.

Example 2.2 The relational structures generated by the import of class Student can be inspected as follows.

```
javai -system Sybase -server java_store -base java_base
      -user john -passwd pj -dumpsq1
```

```
create table Person (
  oid OID,
  primary key (oid),
  class CLASS,
  age int,
  name STRING,
  spouse_oid REF,
  spouse_class CLASS,
  foreign key (spouse_oid) references Person,
  children_oid ARRAY,
  children_class CLASS,
```

```

        foreign key (children_oid) references Person_A
    )
create table Student (
    oid OID,
    primary key (oid),
    responsible_oid REF,
    responsible_class CLASS,
    foreign key (responsible_oid) references Person
)
create table Person_A (
    oid OID,
    primary key (oid),
    size INT
)
create table Person_A_E (
    oid REF,
    foreign key (oid) references Person_A,
    position INT,
    primary key (oid,position),
    element_oid REF,
    element_class CLASS,
    foreign key (element_oid) references Person
)

```

□

Extra database structures, e.g. cursors, sequences, are generated according to the underlying database system. They are used by the generated stored procedures and are made transparent to the runtime system.

3 API

The PJ API consists of a set of classes grouped in the `database.api` package. These are the only classes the user needs to know to be able to write his/her PJ applications. The API classes provide a uniform model for different underlying database models, thereby insulating the application programmer from the syntactic and semantical specificities of different underlying systems. This is in a sense what is already provided by an interface in the style of JDBC, but the PJ API is a higher level interface on top of JDBC, as it hides the relational dimension of stored data by mapping them into Java objects in a transparent way.

The main API classes are `Database`, `Extent` and `Transaction`.

Class `Database` provide methods for establishing database connections, user authentication, checking authorization, opening and closing databases, storing and retrieving objects, security management (users, grants and revokes of rights).

Class `Extent` allows class extents to be scanned and selective retrieval of objects. A class extent contains all the stored instances of a given class.

Class `Transaction` implements a simple flat transaction model assuring the ACID properties [5]. Once a database is opened, a read-only transaction is implicitly started. No locks are hold on objects accessed in a read-only transaction. A transaction must be explicitly started if updates to the database are to be performed. Exclusive locks are hold on objects accessed/updated in a read/write transaction. Such a transaction can be be aborted or committed. Commits can be performed through methods `commit` or `validate` of class `Transaction`. The former clears the database cache whereas the latter keeps loaded objects in the cache.

Example 3.1 The example below sketches a typical database session.

```
database = new Database(); // Create a new database object.
database.connect(...); // Open a connection to a server and perform authentication.
database.open("base1"); // Open a PJ database.
... // Read objects stored in the database in a read-only mode.
transaction = new Transaction(); // Create a new transaction object.
transaction.begin(); // Start a new transaction.
... // create new objects in the database
... // update objects in the database
... // delete objects from the database
transaction.commit(); // Commit updates in the database.
database.close(); // Close the currently active base.
database.open("base2"); // Open another PJ database.
... // Work on objects in base2.
database.close(); // Close the currently active base.
database.disconnect(); //Disconnect from the database server.
```

□

3.1 Persistence management

An object can become persistent only if it is an instance of an imported class. Persistent objects can be inserted into the database, loaded from the database, modified in the database and deleted from the database.

Imported classes must implement the interface `PersistentObject`. This interface declares a set of methods, e.g. `update`, `access` and `delete`, whose implementations are automatically generated by the import program.

The runtime performs *lazy* reading and writing of objects. When an object is updated in the database by applying method `update` to it, the objects it references are not automatically updated. Instead, if referenced objects are to be stored/updated, they must be also explicitly updated through method `update`.

Respectively, when an object is accessed in the database through the method `access`, referenced objects are not loaded, but a *shadow* object is generated for each referenced object instead (unless the referenced object has been previously loaded into the database cache). The method `access` must be further applied on referenced shadow objects if their variables are to be accessed.

An object in a Persistent Java application is in one of the following states:

- **transient**

A transient object is an object that has not been accessed from nor updated in the database and that exists only in the application program heap. It is therefore not in the database cache and has no corresponding object stored in the underlying database.

- **persistent**

A persistent object is an object that has been accessed from the database in the current transaction or in a previous transaction ended with the method `validate`. It has an associated entry in the database cache and is therefore linked to an object stored in the underlying database.

- **shadow**

A shadow object is an object that has been partially loaded from the database when a persistent object pointing to it was loaded. It corresponds to a persistent object in the database and has a corresponding entry in the database cache. Variables of a shadow object are not loaded from the database. A shadow object must be explicitly loaded, by applying method `access`, if its variables are to be accessed.

Example 3.2 The example below shows different state transitions that are detailed in the following sections.

```
...           // p is accessed through another object and is a shadow object.
p.access();   // p is accessed in read-only transaction (not locked).
...
p.access();   // p is not reloaded, it is already in the cache.
...
transaction = new Transaction();
transaction.begin(); // Start a transaction (p becomes a shadow object).
...
p.access();    // p is reloaded, locked and flagged as persistent in the cache.
p.update();    // p will be written at commit time.
if(...)
    transaction.commit(); // p is updated in the database.
else
    transaction.abort();  // p is not updated.

// Cache is cleared, p is no more in the database cache.

transaction.begin(); // Start a new transaction.
...                 // Access p through the class extent.
p.update();
transaction.validate(); // p is persistent in the database cache.
...
p.access();          // p is not reloaded.
...
```

```
transaction.begin();    // Start a new transaction (p is shadow in the cache).
p.access();            // Reload p before updating it (p is locked).
transaction.validate(); // p is unlocked and kept as persistent in the cache.
```

□

As illustrated above, when a transaction starts, all persistent objects found in the database cache, if any, become *shadow* objects. Objects loaded in a previous transaction must then be reloaded from the database inside a transaction if they are to be updated so as to avoid inconsistencies (lost updates and dirty reads).

Inserting new persistent objects in the database

The insertion of a transient object into the database is achieved by applying the method `update` on it. The object is flagged as updated and is written to the database at commit time. It is then automatically added to the corresponding class extent. Transient objects pointed to by a newly inserted persistent object, however, do not become automatically persistent and must be explicitly updated through method `update`.

Example 3.3 The following example writes an instance of `Person` to the database.

```
transaction.begin();
Person John = new Person("John"), Mary = new Person("Mary");
John.spouse = Mary;    // John and Mary are transient objects.
John.update();        // John is flagged as updated in the cache.
transaction.commit(); // John is written to the database.
// Mary is not written and John.spouse is set to null in the database.
```

□

System-supplied methods `access` and `update` can be used in the implementation of specific `deepAccess` and `deepUpdate` methods if one wants objects to be retrieved from and/or written to the database along with (some of) the objects they refer to. Basic methods can be applied recursively on nested objects to allow the storage and retrieval of particular complex object structures. This technique is employed in the implementation of the catalog classes used by the `javai` program.

Storing array variables

The storage of an array is similar to that of a single reference variable. A transient array referred to by a persistent object does not become automatically persistent. To become persistent, the array must be explicitly updated with the static method `update` of class `Database`.

For an array of primitive types, all elements of the array become persistent when the array becomes persistent. For an array of objects, however, a reference to the array object itself is stored within the array variable of the updated object, but array elements are not written themselves. They must be explicitly updated, as for single reference variables. At commit time, if an array element is a transient object, the element is set to null.

Retrieving persistent objects

Stored objects can be accessed through some data entry points defined in the database. Default data entry points correspond to *class extents*. User defined data entry points correspond to *static variables*.

A class extent contains all the instances of a class that have been explicitly updated and written to the database at commit. There are two kinds of class extents: the *proper class extent*, which contains only the instances of the class, and the *transitive class extent*, which contains all instances of the class and its subclasses.

The class `Extent` is provided to allow retrieval of class extents for imported classes. A class extent can be filtered through a predicate. The syntax of a selection predicate is similar to that of the `where` clause of a `select-from-where` SQL query. Extra facilities allow the retrieval of associations between objects.

Example 3.4 The example below illustrate different facilities provided by class `Extent`.

```
Extent personExtent, johnStudents;
Person p, John;
personExtent= Extent.all("Person"); // Retrieve the transitive extent of Person.
...
personExtent = Extent.proper("Person"); // Retrieve the proper extent of Person.
for (Enumeration e = personExtent.elements() ; e.hasMoreElements() ;) {
    p = (Person)e.nextElement();
    ...
}
// Select a particular subset of instances of Person and its subclasses:
personExtent = Extent.all("Person").where("age < 20");
// Select a particular object pointing to another object:
John = Extent.all("Person").where("name = 'John'").element();
johnStudents = Extent.all("Student").where(John.pointedToBy("responsible"));
```

□

Static variables provide a means for the user to define application dependent roots of persistence. If a given static variable is imported into the database, its value can be directly accessed. The static methods `accessStatic` and `updateStatic` are generated by the import program. These methods allow the retrieval and the storage of imported static variables from/into the database.

Example 3.5 The example below assumes the existence of the static variable `Adam` of type `Person` in the class `Person`. By calling the static `updateStatic` method, the reference to the pointed object is stored in the currently active database. The pointed object itself must be explicitly updated, as before.

```
Person.Adam = new Person("Adam");
Person.Adam.update(); // Add Adam to the extent Person in the database.
Person.updateStatic();// Attach Adam to the root of persistence Person.Adam.
```

□

Example 3.6 The retrieval of static variables is performed through the static method `accessStatic`, as illustrated below.

```
Person.accessStatic();
Person adam = (Person)Person.Adam.access();
Person eve = (Person)Person.Adam.spouse.access();
```

□

Once an object has been retrieved, i.e. loaded, from the database, through a class extent or a persistent static variable, objects pointed by it can be directly retrieved. When an object is accessed inside a transaction, a lock is hold on it in the database. After it has been accessed, an object is kept in the cache until a commit or an abort takes place.

Example 3.7 The example below illustrates the update of an instance of `Person`.

```
Person John;
...
John.access(); // John is loaded from the database in a read-only transaction.
System.out.println(John.age);
...
John.access(); // John is already loaded in the cache: does nothing.
...
transaction.begin();
John.access(); // John is reloaded and locked in order to ensure consistency.
John.spouse = Mary;
John.update();
transaction.commit();
```

□

When retrieving a given object, if an object it points to has been already loaded into the database cache, the object being loaded simply points to it through the corresponding reference variable. If, however, the pointed object has not been loaded, a *shadow object* is created for it. The variables of a shadow object are not loaded until the object is explicitly accessed through method `access`, which is generated in each imported class by the import program.

For arrays, when an object containing an array variable is accessed, the array is treated as a reference variable and a *shadow array* is created for it. To access the array, the static method `access` of class `Database` must be used. For an array of primitive types, all elements of the array are loaded from the database when the array itself is accessed. For an array of objects, elements are loaded as shadow objects. Array elements must then be explicitly accessed if their variables are to be accessed.

Elements of an array can be accessed without accessing the whole array. If one wants to update an array rather than its elements, i.e. if elements of the array should be inserted, replaced and/or removed from the array, then the array itself must be accessed before being updated.

Example 3.8 The code below reads an instance of `Person` from the database and illustrates how an array variable and its corresponding elements can be accessed and updated.

```
Person John = Extent.proper("Person").where("name = 'John'").element();
Person[] children = John.children; // children is a shadow array.
Person first = (Person)Database.access(children,0); // Access an array element.
...
first.update(); // Update an array element.
Database.access(children); // Array children is no longer shadow.
for(i=0;i < children.length ;i++)
    children[i].access(); // All children elements are accessed.
...
children[0] = p; // Array children is modified.
Database.update(children); Array children is updated.
```

□

Updating persistent objects

After a Java object has been written to the database, it can be modified in the same transaction or in other transactions. When a transaction commits or aborts, the application must re-access persistent objects before updating them, so as to hold a lock on the accessed object. If the `update` method is applied to an object outside a transaction, an exception is raised. An attempt to update a shadow object also raises an exception.

Example 3.9 The example below illustrates the update of persistent objects. It uses `deepAccess` and `update` methods, as suggested above.

```
Person Adam;
...
transaction.begin();
Adam.deepAccess(); // Loads Adam and pointed objects from the database.
...
Adam.name="Adam Smith";
Adam.spouse.name="Eve Smith"; // Adam.spouse was loaded in method deepAccess.
Adam.deepUpdate(); // Update Adam and pointed objects.
transaction.commit();
```

□

Deleting persistent objects

A persistent object can be deleted from the database by applying the method `delete` to it. If the object is transient, an exception is raised. If the object is shadow or persistent, it is deleted, provided there is no other stored object pointing to it in the database. Otherwise, an exception is raised. This is assured by the referential integrity enforcement mechanisms of the underlying database. After being deleted

from the database, an object becomes transient in the application memory space. If, for instance, a deleted object is further updated and the transaction commits, a new object is inserted for it in the database, as for any transient object.

Deferred versus immediate updates

Contrary to updates to the database, that are deferred to commit time, object deletion is performed immediately in the database when method `delete` is applied to an object.

A special method, namely `writeObject`, is provided in class `Database` to allow immediate, i.e. synchronous, updates to the database. This is particularly useful when references to an object to be deleted must be removed before deleting the object. If the pointing objects are not immediately updated in the database, references to the object to be deleted will not be removed until a commit takes place and the deletion would be aborted. This would force the user to use two transactions: one to cut references to the object to be deleted and another to delete the object itself. With `writeObject`, the update is performed immediately and this restriction can be overcome.

An additional method, namely `destroy`, is provided to allow the application programmer to remove a given object from the database cache (not from the database) before committing or aborting. This allows objects no longer necessary in the current transaction to be garbage collected and can be used to optimize memory space usage. Destroyed objects that were locked in the database are nevertheless not unlocked until a commit or an abort takes place.

More in general, the `writeObject` method can be used in combination with the `destroy` method to distribute database updates across a transaction and reduce the commit response time.

4 Implementation

The implementation of Persistent Java uses standard Java classes where appropriate and builds on the style and virtues of the existing core Java classes [3].

The interface is implemented with the standard Java language. No changes to the Java compiler nor to the Java virtual machine were attempted in the current version of Persistent Java. The amount of C code used in the implementation of PJ is reduced to a minimum, as calls from Java to native C code have a number of drawbacks in the security and automatic portability of applications.

Errors and exceptions raised by the underlying database are captured by the runtime system and transmitted to the Java application program as a Java exception.

Some facilities of the underlying database systems were used in the implementation of the runtime interface. For instance, the enforcement of referential integrity constraints provided by relational systems relies on the use of primary and foreign keys. References between objects are implemented as foreign keys in the underlying database and object identifiers are modeled as system generated primary keys in the corresponding tables.

The performance of Persistent Java can be tuned by the user to meet particular application needs. These facilities should be in general not necessary, but can be applied in data intensive applications with special space and/or time related requirements. The configuration file (e.g. redefinition of default type mapping for storage of arbitrarily long raw data), together with some special system generated methods, can be used to that end.

4.1 Database cache

The runtime keeps a database cache relating Java application objects to objects stored in the underlying database. The database cache is used to allow persistent objects to be manipulated by a Java application in a transparent way.

The database cache is implemented by a set of hashtables. These tables maintain the correspondence between oids of database objects and addresses of Java objects in the application heap.

Objects in a Persistent Java application can be in a number of different states, namely *transient*, *persistent*, *updated* and *shadow*. The state transition is driven by the application of PJ API methods on objects and the runtime keeps track of object states so as to perform the corresponding actions accordingly.

The hashtables composing the database cache are inspected and/or updated when methods `access`, `update`, `delete` and `destroy` are applied to Java objects. Such tables are implemented as instances of the class `java.util.Hashtable`. When a database is opened and after an abort or a commit, these hashtables are empty.

4.1.1 Table cacheToBase

This table associates to an object in the application memory space an object in the database. The information stored in this table is composed of: (1) a reference to the persistent object (the hash key); (2) the database object identifier (oid); (3) a flag indicating the object status; (4) the last transaction where the object has been accessed.

The possible values for the status flag are:

- **I**: the object must be inserted into the database at commit;
- **U**: the object must be updated in the database at commit;
- **S**: the object is a shadow object;
- **P**: the object is persistent and is not updated at commit;

As discussed in Section 3, an object is always in one and only one of the following states at a given time:

- **transient**

The object has not been accessed/updated in the database and is not in the cache nor in the database.

- **persistent**

The object has been accessed/updated in the database and is kept in the cache. Possible values for its flag in the cache are **I**, **U** and **P**. The action undertaken at commit depends on the status flag. If it is **I**, the object was transient and was updated through **update** in the current transaction. It is therefore inserted into the database at commit. If the flag is **U**, the object was already persistent and was accessed through **access** followed by an **update** in the current transaction and is therefore written to the database at commit. Finally, if it is **P**, the object is stored in the database and was accessed through **access** but was not updated in the current transaction and is therefore not written to the database at commit.

- **shadow**

The object has been partially accessed in the database as a result of an explicit **access** to an object referencing it. The object is in a shadow state in the cache, i.e. its flag is set to **S** in the table **cacheToBase**.

If an object has been accessed in a transaction previous to the current transaction (and kept in the cache after a **validate**), its transaction identifier, in the cache, is different from the current transaction identifier kept by the runtime. This object is also considered as a shadow object by the runtime.

4.1.2 Table **baseToCache**

This table associates, to a given object in the database, an object in the application memory space. It is used to allow persistent objects to be shared. When a reference to a persistent object is read from the database as a foreign key value, if the corresponding oid is a key in this table, the object associated to this key is assigned to the corresponding reference variable of the referencing object being loaded. Otherwise, a new shadow object is created and assigned to that variable. The class of the shadow object is determined by inspecting the value of the **class** column associated to the foreign key (see Example 2.2).

4.2 Writing objects

As discussed above, an object becomes persistent when the method **update** is applied to it. The **cacheToBase** hashtable is consulted and different actions are undertaken according to the status of the object.

If the object is not found in the table **cacheToBase**, it is a transient object. In this case, an entry in the table **cacheToBase** is generated for it, with the status set to **I** and an undefined oid.

If the object is found in the cache, the following happens:

- If **Flag = I** or **Flag = U**: nothing is done (object is already flagged and will be inserted or updated).
- If **Flag = P**: the flag is set to **U** (object will be updated).

- If `Flag = S`: an exception is raised.

All object insertions and updates in the database are performed only at commit, when the `cacheToBase` table is scanned. Objects in this table are inserted or updated in the database, through the corresponding `insert` or `update` stored procedure, according to their status flag.

Object insertion

For each object in `cacheToBase` whose flag is `I`, the corresponding row(s) is (are) created in the database. This is achieved by calling the `insert` stored procedure generated by the import program for the class of the object being inserted.

The values of the variables of primitive type and string type are passed as parameters to the corresponding `insert` stored procedure. This procedure inserts a new row in the corresponding table in the underlying database. For reference variables, the `cacheToBase` table is consulted. If the referred object is found and its database oid is well-defined (i.e. if its flag is not `I`), the oid is passed as a parameter to the stored procedure to compose the foreign key in the row being inserted for the referencing object. If the referred object is not found, a reference to `null` is passed as a parameter for the corresponding foreign key.

Finally, if the referred object is found and its flag is `I`, a *shadow row* is created for this object in the corresponding table to allow the retrieval of its database oid. Its flag is set to `U` and the returned oid is used to compose the foreign key of the referencing object and to update the oid of the referred object in the cache. When this pointed object is further reached during the scan of the cache, the corresponding shadow row is updated and loses its shadow status.

After being inserted in the database at commit, the flag of an object is set to `P`.

Object update

For each object in `cacheToBase` whose flag is `U`, the corresponding database row(s) is (are) updated. This is achieved by calling the `update` stored procedure generated by the import program for the class of the object being updated.

The procedure for update is very similar to that defined for object insertion, except that, for update, the oid of the object is already known and is passed as a parameter to the `update` procedure to allow the selection of the right row(s) to be updated in the database, whereas the insertion procedure is in charge of generating and returning the oid of the object being inserted. The `update` stored procedure returns no value.

Example 4.1 We show below the update stored procedure generated for class `Student`, which calls the update procedure generated for the superclass `Person` of example 2.1. The syntax used in this example is that of Sybase Transact-SQL.

```
create procedure update_Student @oid REF, @age int,
                               @name varchar(255),
                               @spouse_oid REF, @spouse_class CLASS,
```

```

                                @children_oid REF,@children_class CLASS,
                                @responsible_oid REF,@responsible_class CLASS
as   execute update_Person @oid,@age,@name,@spouse_oid,@spouse_class,
                                @children_oid,@children_class
      update Student set
          responsible_class = @responsible_class,
          responsible_oid = @responsible_oid
      where @oid = oid

```

□

4.3 Reading objects

In order to access a persistent object, the `access` method must be called on this object. The `cacheToBase` hashtable is consulted to read the object status, and different actions are undertaken according to the state of the object being read, as depicted below.

If the object is not found in the cache, it is a transient object and an exception is raised. If it is found in the cache with the status flag set to `P`, `I` or `U`, nothing is done, as the object has already been accessed.

If the object is found in the cache with flag `S` or its associated transaction identifier is lower than the current transaction identifier, the object is in a shadow state and will be therefore accessed in the database. Its status is set to `P` and all its primitive and string type variables are loaded from the database through the corresponding `select` stored procedure. The database `oid` of the shadow object being read, stored in the `cacheToBase` hashtable, is passed as a parameter to the `select` procedure. This procedure returns the primitive type values, the string values and the references to other objects (stored as foreign keys in the relational schema).

For a reference variable `v`, the values of columns `v_class` and `v_oid` are used as a key to access the corresponding object in the table `baseToCache`. If an entry for this key already exists, the referred (possibly shadow) object is already in the cache and a reference to it is assigned to the variable `v` of the object being accessed. The flag of the referred object is not changed. If, on the other hand, the referred object is not found in the cache, a corresponding shadow object is created and inserted into the cache with its flag set to `S`. A reference to this newly created shadow object is assigned to the variable `v` of the object being accessed.

Objects accessed through a class extent are persistent and have their flag set to `P` in the cache. When `personExtent = Extent.all("Person")` is executed, for instance, the following query is run on the underlying database:

```
select oid+class from Student
```

where `oid+class` stands for the concatenation of the `oid` and `class` column values. The retrieved oids are kept by the `personExtent` object. Objects corresponding to the retrieved oids are accessed on demand, as the extent is scanned. The shadow instances are created based on the `class` information.

Example 4.2 We show below the select stored procedure generated for class `Student`, which retrieves data allowing a shadow object to be completely loaded from the database.

```
create procedure select_Student
    @oid REF, @class CLASS output,
    @age int output,@name varchar(255) output,
    @spouse_oid REF output,@spouse_class CLASS output,
    @children_oid REF output,@children_class CLASS output,
    @responsible_oid REF output,@responsible_class CLASS output
as select @class=class,@age=age,@name=name,
    @spouse_oid=spouse_oid,@spouse_class=spouse_class,
    @children_oid=children_oid,@children_class=children_class,
    @responsible_oid=responsible_oid,@responsible_class=responsible_class
from Person,Student
where @oid = Person.oid and
    @oid = Student.oid
```

□

4.4 Deleting objects

Object deletions are performed when method `delete` is applied on a given persistent or shadow object. After checking the state of the object, the corresponding `delete` stored procedure is called. The oid of the object to be deleted is passed as a parameter to the `delete` procedure. If the row corresponding to the oid is referred to by other rows through foreign keys in the database, this row is not deleted from its table. This is assured by the referential integrity enforcement mechanisms of the underlying database. An exception is raised by the runtime in this case. If the row is not referred to by other rows, it is removed from its table and the corresponding object is as a consequence, deleted from the database and the object is removed from the cache. The object becomes transient.

4.5 JDBC

As shown in Figure 1, the PJ runtime is implemented on top of a JDBC compliant interface layer. This layer is not a JDBC driver, in that it does not implement all the functionalities of a driver as described in the JDBC specification, but it is compliant to JDBC and it should be therefore easy to replace it by a JDBC driver if one were available. This promotes portability, as Persistent Java applications would in principle run on top of any database for which a JDBC driver were available, provided the driver and the corresponding underlying database implement the functionalities we rely upon.

The C language is used to implement the native methods of classes in package `database.jdbc` and its subpackages (`oracle` and `sybase` for Unix and NT platforms are available for the moment). This is the only part of Persistent Java that is not implemented in Java².

²We exclude the stored procedures, that are system generated.

5 Future work

Although some changes to the compiler could have improved the performance and transparency of the API, we preferred to rely on the existing compiler and virtual machine for the implementation of the current database interface. Migration of part of the interface to the core of the Java runtime system might be attempted after some experimentation with the current interface.

In particular, we are interested in the three following directions:

- Bytecode post-processing to provide persistence in a transparent way.
Calls to `access` and `update` methods can be inserted into a persistent capable class bytecode wherever a persistent object is de-referenced and to implement persistence by reachability in a transparent way. This would free the programmer from having to explicitly call this methods from his/her applications.

- Use of the set of currently implemented primitives to provide persistence on top of a generic database for which a data mapping mechanism is available.

The implementation of PJ on top of the O₂DBMS [1] can build on the current runtime implementation by just mapping Java objects into the O₂ data model. This can be further generalized to make the mapping and database interface ODMG compliant, in which case only specific drivers should be provided by different ODMG compliant database constructors to allow PJ applications to run on top of their respective products.

- Provide support for exporting legacy databases structure and data as Java classes and respective instances.

By relying on the same runtime, we have implemented a preliminary prototype that generates a set of Java classes from an existing relational database schema and that allows relational data to be manipulated as Java objects.

6 Conclusion

We presented an overview of the main features of Persistent Java and we briefly discussed some implementation issues. Due to space limitations, we were not able to give more details on the Persistent Java design and implementation.

Persistent Java is currently implemented for Oracle and Sybase database systems running on Solaris and NT platforms. It is available as a library of Java classes, specific plataform and system dynamic lybraries or DLL's and general auxilliary tools, e.g. `javai`, `create_base`, `create_importer`.

The current API and runtime implementation will serve as a basis for the short term implementation of the different products mentionned in Section 5.

References

- [1] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O₂*. Morgan Kaufmann, 1992.
- [2] R.G.G. Cattell, editor. *Object Database Standard : ODMG - 93, Release 1.2*. Morgan Kaufmann, San Francisco, 1996.
- [3] David Flanagan. *Java in a Nutshell*. O'Reilly and Associates, Inc., 1996.
- [4] Graham Hamilton and Rick Cattell. *JDBC: A Java SQL API*, June 1996.
- [5] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [6] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham, and S. Dao. Construction of a relational front-end for object-oriented database systems. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 476–483, Vienna, Austria, April 1993. IEEE Computer Society Press, Washington, DC.
- [7] J. E. Rumbaugh. Relational Database Design Using an Object-Oriented Methodology. *Comm. of the ACM*, 31(4):414–427, 1988.