mcobject®
precision | data management

Perst, a Simple, Fast, Convenient Database for Java and .NET

Perst®

Introduction and Tutorial
October 10, 2012

22525 SE 64th Place, Suite 302
Issaquah, WA 98027 USA
Phone: 425-888-8505
Fax: 425-888-8508
E-mail: info@mcobject.com
www.mcobject.com

1. Overview

1.1. What is Perst?

Perst is a pure Java/.NET/Mono object-oriented embedded database system. *Object-oriented* here means that Perst is able to store/load objects directly. *Embedded* means Perst is intended to be included inside an application that needs its own data storage.

Perst's goal is to provide programmers with a convenient and powerful mechanism to deal with large volumes of data. Several fundamental assumptions determine Perst's design:

1. Persistent objects should be accessed in almost the same way as transient objects.
2. A database engine should be able to efficiently manage much more data than can fit in main memory.
3. No specialized preprocessors, enhancers, compilers, virtual machines or other tools should be required to use the database and to develop applications using it.

1.2. Database engine

Perst is distributed as single .jar fileTo use it in an application, the only requirement is to include it in the classpath. The Perst distribution includes several versions of the library:

perst.jar
: Perst Java version for Java JDK 1.5 and higher

perst14.jar
: Perst Java version for Java JDK 1.4

perst11.jar
: Perst.Lite Java version for JDK 1.1.*

perst-rms.jar
: Perst.Lite Java version for J2ME (MIDP 1.0/CLDC 1.1)

perst-jsr75.jar
: Perst.Lite Java version for J2ME with optional JSR-75 package (MIDP 1.0/CLDC 1.1/JSR-75)

1.3. Structure of packages

The Perst database engine consists of two main packages, `org.garret.perst` and `org.garret.perst.impl`. The programmer mostly uses interfaces and classes from the `org.garret.perst` package, while the `impl` package provides implementations of these classes and interfaces. This structure allows easy changes to the implementation of a particular class without affecting applications using it.

From the application's point of view, Perst consists of a Persistent class that is the base class for all persistence-capable objects and provides methods for loading/locking and

storing objects; various collections classes (indexes) to provide fast access to objects; and a Storage class that is both responsible for transaction management and used as a "factory" for index classes.

The C# version of Perst was produced from the Java version using the Java-to-C# converter provided by Microsoft Visual Studio. Additional changes to the .NET version of Perst enabled support of enums, properties, unsigned types, decimal/guid/DateTime data types, and dynamic generation of pack/unpack methods, among other things. To gain compatibility with the .NET code style guide, the naming convention was changed for all Perst methods.

As a result, Perst now provides separate Java and .NET sources trees. No automated procedure exists to propagate changes in both source trees. Also, since Java has no built-in preprocessor to choose between compilation paths, the Java version of Perst offers three different source trees, for Java 1.4, Java 5.0 and for Perst Lite (Perst Lite is a sub-edition of the database that supports Sun's Java ME, also known as Java 2, Micro Edition, or as J2ME). This Tutorial primarily uses Java 1.4 examples, and sometimes also Java 5.0 generic classes. The most accurate information about Perst APIs can be found in its documentation, available for free download from McObject's Web site (follow the link from www.mcobject.com/perst).

2. Storing objects in the database

2.1. Open store

Let's start developing a Perst application. Perst's primary goal is to allow the programmer to work with persistent objects in a manner as similar as possible to working with normal transient objects. However, persistence has certain aspects that cannot be completely hidden from the programmer. For example, a Perst "storage" (file) where objects will be stored must be specified, and data consistency must be provided for, in case of faults. With Perst, these tasks are performed by the `Storage` class. This class is abstract and the programmer should use the `StorageFactory` class to create a storage instance:

```
// get instance of the storage
Storage db = StorageFactory.getInstance().createStorage();
```

Once this storage instance is created, it can be opened:

```
// open the database
db.open("test.dbs", pagePoolSize);
```

In the example above, the first parameter specifies the path to the database file, and the second parameter sets the size of the page pool. The page pool is used by Perst to keep the most frequently used database pages in memory, which reduces disk IO and improves performance. Generally, a larger page pool leads to faster execution. However, memory is also needed for other applications and for the operating system; if you specify a very large page pool and it doesn't fit in main memory, it will be swapped to disk and

performance will degrade. Therefore, system resources must be considered when specifying the page pool size.

In addition to specifying the path to the database file, it is possible to pass details of the storage implementation to the `IFile` interface. This interface provides some special file implementations, such as compressed file, encrypted file and multifile (a virtual file consisting of several physical segments). The programmer can also provide his or her own implementations of the `IFile` interface allowing Perst to work with specific media or data sources.

It is possible to use Perst as an in-memory database, in which all data is stored and managed in main memory. This requires using the `NullFile` stub class and specifying `INFINITE_PAGE_POOL` as the page pool size. In this case, the page pool will be extended on demand:

```
db.open(new NullFile(), // Dummy implementation of IFile interface
        Storage.INFINITE_PAGE_POOL); // page pool is extended on demand
```

When the application has finished using a Perst database, the database should be closed using the `Storage.close()` method:

```
        // get instance of the storage
        Storage db = StorageFactory.getInstance().createStorage();
        db.open("test.dbs", pagePoolSize); // Open the database
        // do something with the database
        db.close(); // ... and close it
```

2.2. Root object

After the storage instance is opened, the database can access persistent objects from it. When the application accesses normal (transient) objects, it uses references to these objects kept in program variables or in fields of other objects. But when opening a database containing persistent objects (objects whose "life" is longer than the application session lifetime), the application has no reference to these objects.

A mechanism is needed to get the references, and Perst provides it using something called a *root* object. The storage can have only one root object. It is obtained using the `Storage.getRoot` method, which returns null if the storage is not yet initialized and the root object not yet registered—in that case, it is necessary to create an instance of the root object and register it in the database using the `Storage.setRoot` method. Root objects can be of any persistence-capable class (a concept that is discussed in the next section).

Once a root object has been registered, the `Storage.getRoot` method will always return a reference to this object. And this object can contain references to other persistent objects that can be accessed by the application. So only the root object requires a special access semantic. Accessing all other objects is performed the same way as getting normal (transient) objects.

The code below shows initialization of the database:

```
MyRootClass root = (MyRootClass)db.getRoot();// get storage root
if (root == null) {
    // Root is not yet defined: storage is not initialized
    root = new MyRootClass(db); // create root object
    db.setRoot(root); // register root object
}
```

In most OODBMSs, it is possible to get an object using a string key, or to get a *class extent* (the set of all instances of the particular class). In Perst, such functionality is provided by using an index collection as the root object (indexes are discussed later). The key of this index is a string identifier or class name. And the value associated with the key can be a persistent object or persistent set representing a class extent. Below are examples of typical root class definitions.

In this example, the root object is used as a dictionary, allowing access to persistent objects using a string key:

```
// get instance of the storage
Storage db = StorageFactory.getInstance().createStorage();
// open the database
db.open("test.dbs", pagePoolSize);

Index dictionary = (Index)db.getRoot(); // get storage root
if (root == null) {
    // Root is not yet defined: storage is not initialized
    root = db.createIndex(String.class, // key type
                          true); // unique index
}
// Now we can get persistent object by string identifiers:
MyPersistentClass obj
    = (MyPersistentClass)root.get("main-object");
// and store it in the storage with specified key binding:
obj = new MyPersistentClass();
Root.put("yet-another-object", obj);
```

Here the root object is used as a collection of class extents. Class name is used to obtain a collection of all instances of this class:

```
// get instance of the storage
Storage db = StorageFactory.getInstance().createStorage();
// open the database
db.open("test.dbs", pagePoolSize);

Index dictionary = (Index)db.getRoot(); // get storage root
if (root == null) {
    // Root is not yet defined: storage is not initialized
    root = db.createIndex(String.class, // key type
                          true); // unique index
}
// Now we can get collection of all instances of a class
IPersistentSet classExtent = (IPersistentSet)
```

```
    root.get("com.mycorp.app.MyPersistentClass");
if (classExtent == null) {
    classExtent = db.createSet(); // create class extent
    root.put("com.mycorp.app.MyPersistentClass", classExtent);
}
// iterator through all instance of the class
Iterator i = classExtent.iterator();
while (i.hasNext()) {
    MyPersistentClass obj = (MyPersistentClass)i.next();
    obj.show();
}
// Add newly created objects in corresponding class extent:
MyPersistentClass obj = new MyPersistentClass();
classExtent.add(obj);
```

Here the root object contains references to the various indexes that are needed to search database objects:

```
// There should be one root object in the database, containing
// collections used to access all other objects in the storage.
class MyRootClass extends Persistent
{
    // index on MyPersistentClass.intKey
    public FieldIndex<MyPersistentClass> intKeyIndex;
    // index on MyPersistentClass.strKey
    public FieldIndex<MyPersistentClass> strKeyIndex
    // index on MyPersistentClass, which key doesn't
    // belong to the class
    public Index<MyPersistentClass>      foreignIndex;
    public MyRootClass(Storage db) {
        super(db);
        intKeyIndex = db.<MyPersistentClass>createFieldIndex(
            MyPersistentClass.class, // indexed class
            "intKey", // name of indexed field
            true); // unique index
        strKeyIndex = db.<MyPersistentClass>createFieldIndex(
            MyPersistentClass.class, // indexed class
            "strKey",  // name of indexed field
            false); // index allows duplicates (is not unique)
        foreignIndex = db.<MyPersistentClass>createIndex(
            int.class, // key type
            false); // index allows duplicates (is not unique)
    }

    // Default constructor is needed for Perst to be able to
    // instantiate instances of loaded objects
    public MyRootClass() {}
}

public class AppMainClass extends Persistent
{
    static public void main(String[] args) {
        // get instance of the storage
        Storage db =
            StorageFactory.getInstance().createStorage();
```

```
            // open the database
            db.open("testidx.dbs", pagePoolSize);

            MyRootClass root = (MyRootClass)db.getRoot();
            // get storage root
            if (root == null) {
                // Root is not yet defined: storage not initialized
                root = new MyRootClass(db); // create root object
                db.setRoot(root); // register root object
            }

            // Create new object instance
            MyPersistentClass obj = new MyPersistentClass();
            obj.intKey = 1;
            obj.strKey = "A.B";
            obj.body = "Hello world";

            // ... and insert it in the corresponding indexes
            root.intKeyIndex.put(obj);
            // add object to index on intKey field
            root.strKeyIndex.put(obj);
            // add object to index in strKey field
            root.foreignIndex.put(new Key(1001), obj);
            ...
            db.close(); // close the storage
        }
    }
```

2.3. Persistence-capable classes

Perst provides orthogonal persistence - i.e. any database class is persistence-capable (may be stored in the database). But to achieve the best performance and flexibility, it is preferable for any persistence-capable class to implement `IPersistent` interface.

The default implementation of this interface is the `org.garret.perst.Persistent` class. In most cases, persistence-capable classes should be derived from the `Persistent` base class (or from some other persistence-capable class). An alternative implementation of the `IPersistent` interface may be needed if application classes must be derived from some other external class for which sources are not available. In such cases, the solution is to provide an implementation derived from the required base class and reuse `Persistent.java` code to implement methods of the `IPersistent` interface.

The following example shows the derivation of a persistence-capable class:

```
// All persistence-capable classes should be derived from
// Persistent base class
class MyPersistentClass  extends Persistent
{
    public int intKey;    // integer key
    public String strKey; // string key
    public String body;   // non-indexed field

    public String toString() {
        return intKey + ":" + strKey + ":" + body;
```

```
        }
    }
```

But what *is* a persistence-capable class? Why not just a *persistent* class? The following section explains.

2.4. Persistence by reachability

Once the root object is obtained, how are new objects to be saved in the database? One solution is to use the `store` method in the `Persistent` class.

But Perst provides a more convenient model which is called *persistence by reachability*. Its main goal, once again, is to provide transparent persistence: the programmer should not have to worry about where an object is to be stored, or about storing the object at all. In Java applications, objects have different lifetimes. Some are used only temporarily and are reclaimed by the garbage collector soon after their creation. Others (for example, an object representing a HTTP session) have longer lifetimes. A persistent object should survive termination of the application session and be available for activation when the application is started again.

And clearly, an object's lifetime is determined by references to the object. If an object is referenced only by local variables or parameters, (as in the expression `log.write(obj.toString())`), then the garbage collector is able to reclaim that object when control is returned from this method. However, if the object is referenced from some other object or from a static field, then the object will be present in memory as long as such references exist. If the object is referenced from the field of a persistent object, then the referenced object should also be persistent (otherwise this reference will be invalid after the application restarts).

This simple idea is *persistence by reachability*. To become persistent, an object needs to be referenced from some other persistent object. The only exception is the root object, which becomes persistent when it is registered as the storage root.

In Perst, persistence by reachability is leveraged by using the database's `Persistent.modify()` method. Consider a persistence-capable class `Employee` and a persistent instance of this class `e1` that is already stored in the database. Next, create a new instance `t1` of a persistence-capable class `Task` and assign it to the employee by setting a reference to this object from the corresponding field of `e1`. Now `t1` should also become persistent because it is referenced from persistent object `e1`. But how does the database engine know that object `e1` was modified? The database system is informed about it using `Persistent.modify()`. (Note that modify should be invoked for `e1`, not for `t1`. Invocation of the method for the newly created `Task` will have no effect, because a newly created object is always considered to be modified.) Failure to invoke the modify method can result in unpredictable application behavior. The object may be saved (because modify is called from some other place), or the modification may be lost.

For the following reasons, the `modify()` method is often better than the `store()` method for saving objects in the database:

1. `modify()` may only have to be invoked rarely: if several objects are created and assigned to components of some persistent object, modify need only be invoked once for this object.
2. By using `modify()` instead of `store()`, it may be possible to reduce the number of times an object is written to the storage. The object can be modified multiple times but saved to the storage only once. However, with the `store()` method, it is saved each time the method is invoked.
3. In many cases, the programmer need not explicitly invoke the `modify()` method, because newly created objects are inserted in some Perst collection class implementations that track modifications themselves.

Below is an example of storing an object:

```
// Create instance of the persistence-capable class.
MyPersistentClass obj = new MyPersistentClass();
obj.intKey = 1;
obj.strKey = "A.B";
obj.body = "Hello world";

root.intKeyIndex.put(obj);
// add object to index on intKey field
root.strKeyIndex.put(obj);
// add object to index in strKey field
```

Persistent objects will be present in the storage until explicitly de-allocated using the `deallocate` method defined in the `Persistent` class, or until they are implicitly de-allocated by the Perst garbage collector. With Perst, using garbage collection is optional. Explicit memory de-allocation is faster but lacks protection against bugs such as *dangling references* or *memory leaks*.

Perst garbage collection can be started explicitly using the `Storage.gc` method, or started automatically by specifying an amount of allocated objects as a threshold. The garbage collector de-allocates all persistent objects that are not reachable from the root object. Even when explicit memory de-allocation is preferred, it is possible to use the garbage collector to ensure there are no memory leaks in the database.

Example of object de-allocation:

```
Iterator i = root.intIndex.iterator();
// iterate through all objects
while (i.hasNext()) {
    MyPersistentClass obj = (MyPersistentClass)i.next();
    i.remove(); // exclude object from index
    obj.deallocate(); // deallocate object
}
```

All the examples above illustrate how to store/update/delete persistent objects whose class implements the `IPeristent` interface. As previously mentioned, Perst supports

orthogonal persistence and allows instances of any class to be stored in the database. Certainly, such a class will not have such methods as `modify()`, `store()`, `deallocate()` or `load()`. Instead, methods with the same names as those used in the `Storage` interface should be used, with the object passed as a parameter:

```
class MyClass // This class is not derived from Persistent

{
    public int intKey;    // integer key
    public String strKey; // string key
    public String body;   // non-indexed field
}

...

// Create instance of the class.
MyClass obj = new MyClass();
obj.intKey = 1;
obj.strKey = "A.B";
obj.body = "Hello world";

// Insert object in indices.
root.intKeyIndex.put(obj); // add object to index on intKey
root.strKeyIndex.put(obj); // add object to index in strKey

...

// If we want to update object,
// we should first remove it from this indices
root.intIndex.remove(obj);
obj.intKey = 2; // assign new value
db.store(obj);  // store updated object in the database
root.intKeyIndex.put(obj); // reinsert object in the index

...

// Delete object.
root.intKeyIndex.remove(obj);// remove object from intKey index
root.strKeyIndex.remove(obj);// remove object from strKey index
db.deallocate(obj); // deallocate object
```

2.5. Relations between objects

The simplest one-to-one relation between objects A and B is represented by assigning a reference to object B in a field of object A. References can also be used to represent one-to-many relationships, with a reference to the relation "owner" stored in the field of relation "members" (for example class `Student` can have field `Tutor tutor` referencing his tutor). An index for this reference field can serve to efficiently locate all members of the relation (given a reference to the tutor, we can locate all his students).
But many-to-many relations must also be represented. And sometimes, even for one-to-many relations, it is more convenient or efficient to keep a list of all relation members

instead of locating them using a query. Perst offers seven (!) different ways to associate one object with multiple objects:

1. Standard Java arrays. This is certainly the most familiar approach for most users. But it has disadvantages, including:
   - All array members must be loaded when the object containing this array is loaded.
   - Adding/removing elements to/from the array is not possible. Instead, a new array must be created and elements copied to it.
   - An array is stored as content of a persistent object, rather than as a separate object, so the same array can't be referenced from different persistent objects. Also, when modifying the content of the array, care must be taken to invoke the `modify()` method for the container object.
   - It is inefficient to have arrays with large numbers of members (greater than 100), since search and update operations will become too costly.
2. Standard Java collection classes (defined in java.util package or System.Collection and System.Collections.Generics namespaces). Perst stores these collections as embedded collections inside persistent objects referencing them. This makes work with small collections more efficient, since collections are loaded together with the object, so there is no need for extra disk read operations. But far as such a collection is not a separate persistent object, it is not assigned an object identifier (OID) and cannot be referenced from more than one persistent object.
3. Perst `Link` class. This is Perst's analogue to standard Java's `ArrayList` class. The main difference is that Perst's `Link` class loads its elements on demand. As an array, a link is not a separate persistent object, but is embedded in a container object. Limitations on the number of members are almost the same as for arrays, since manipulation with large lists requires significant CPU and memory resources.
4. Perst `Relation` class. The only difference between this and the `Link` class is that `Relation` is a persistence-capable class that is stored in the database as a separate object. Therefore, it can be referenced from multiple persistent classes. Internally, `Relation` uses `Link` as its component, so size limitations are the same as for links and arrays.
5. Perst list implementations (`IPersistentList` interface). Implementation of a persistent list is based on a B-Tree index, so it can handle very large relations. Its main drawback is its relatively large space consumption in the case of small relations: even if the relation consists of one element, it will use several kilobytes.
6. Perst set implementations (`IPersistentSet` interface). Persistent set is also implemented using the B-Tree and the main difference from persistent list is that set elements have no fixed position and cannot be accessed by the index. Instead, the program iterates through all set members, checks if the set contains specified elements, and then adds or removes elements. The size overhead is the same as for persistent list.
7. Perst scalable set implementations. This implements the same `IPersistentSet` interface but combines advantages of the array-based

`Link` and the B-Tree-based `PersistentSet`. For small numbers of elements, this class uses the `Link` class to store relation members. When the number of elements exceeds a threshold, a B-Tree is created and used instead of `Link`. The scalable set combines the minimal space overhead of an array with the B-Tree's ability to work with large amounts of data. It is practical when estimating the typical size of a relation is difficult or impossible. The Perst scalable set doesn't provide access to elements by position.

3. Retrieving objects from the database

3.1. Transparent persistence

Assume that a database containing persistent objects is initialized and an application wants to access these objects. As mentioned above, the first step is to obtain the root object using the `Storage.getRoot` method. All other persistent objects can be obtained using references and methods of the root object. But how does this work in practice?

Java doesn't allow detection, overwriting or wrapping the invocation of a method or access to an object field. This suggests that a database engine cannot load an object from storage on demand without using a preprocessor or a specialized virtual machine—in other words, that the programmer will have to load the object explicitly. But the main goal of Perst, and of most other object-oriented database systems, is to provide transparent persistence: to allow the programmer to work with persistent objects in the same way as with normal (transient) objects. If the programmer has to explicitly load an object, transparency seems compromised. However, Perst offers solutions for object-loading that reduce the impact on transparency to a vanishingly small level.

3.2. Implicit recursive object-loading

One solution is recursive loading of all referenced objects. When an object is loaded, the database engine inspects all reference fields of this object and recursively loads all referenced objects.

This requires a mechanism to detect loops and prevent infinite recursion. Each storage has a root object, and all other persistent objects are accessible from it. So calling the `Storage.getRoot` method causes all persistent objects in the database to be loaded into memory. This is usually undesirable, especially when working with large databases that can't fit into main memory. A better solution is the following:

3.3. Eliminating recursion and explicit object-loading

Two options have been presented: make the programmer explicitly load all accessed objects, or recursively load all referenced objects. Both have disadvantages. As it turns out, real transparent loading of objects occurs only when the language supports

behavioral reflection, i.e. the ability for the language to change the behavior of objects. Neither Java nor C# support this. Behavioral reflection can also be emulated using source level or byte code processors (also called *code enhancers*). Perst doesn't contain such a preprocessor, but it does provide integration with popular products such as JAssist and AspectJ that can patch byte code (either statically or at load time) and so achieve transparent activation of objects.

And even without using these tools, Perst provides a compromise solution that eliminates the worst aspects of both explicit loading, and implicit recursive loading. By default, Perst recursively loads all referenced objects. But recursion can be controlled by the programmer. The `recursiveLoading` method in the Persistent class has a very simple implementation: it always returns `true`. To eliminate recursion, the programmer can redefine this method to return `false` instead. When this is done, recursive loading of referenced objects is stopped at the instance of this class, so that components of the class are not implicitly loaded and the implementer of this class has to explicitly load them using the `Persistent.load` method.

At first, this compromise might not seem so helpful for developers, who still have to find a way to eliminate recursion and explicitly load objects. But in practice, it meshes well with object-oriented applications' typical data model, in which persistent data consists of tightly coupled clusters of interrelated objects (groups of objects referencing each other). For example, the object representing a computer in a shop may contain references to the objects describing it: CPU, HDD, memory, monitor, etc. Objects in such a cluster are typically accessed together by the application, using indexes (for example, the computer can be selected by CPU type, by price range, or by amount of memory). Perst's index implementations always stop recursive loading of objects. All Perst indexes load retrieved objects on demand. This means that when such an index is accessed, it will never try to fetch all its members. Instead, members will be loaded only when they are located by iterating through index members or search results.

So in most cases there is no need for the programmer to eliminate recursion and explicitly load objects. A programmer should follow a simple rule: avoid using large arrays and linked lists of persistent objects. Instead, use Perst's collections, which efficiently implement loading of their members on demand. Programmers implementing their own collection class in Perst must remember to eliminate recursive loading and to explicitly load collection members.

4.Searching objects

4.1. Using indexes

Perst's power lies in its varied collection classes for persistent objects, which enable the choice of indexes best suited for particular applications. Standard Java class libraries provide many different collection classes: lists, arrays, maps, trees, etc. But database applications have additional demands for persistent data containers. For example, search methods should support range queries (`select order where shipment date is between 01.01.2008 and 01.12.2008`) and sorting of returned data (`select`

`employee order by salary`).

Collections for persistent data should be able to handle large volumes of data that don't fit in main memory. However, all standard JDK collection classes are optimized on the assumption that all collection members are present in memory. Using such classes when most persistent objects are on disk is very inefficient. Because of such limitations, Perst provides its own collections (which nevertheless implement many basic interfaces from system collections packages).

The most efficient and universal index data structure for persistent data located on disk is the B-Tree. B-Trees consist of large pages (tree nodes) which contains a lot of <key, value> pairs, where key is an index key and value refers to either the child B-Tree page, or to the indexed object. So a B-Tree is a tree with large width and small height. Small height is the key factor of good B-Tree performance for on-disk data. Each operation with a B-Tree (insert, search, remove) requires access to at most H pages where H is the height of the B-Tree (in other words, the complexity of basic operations with a B-Tree is `O(log(N))`, where `N` is number of elements in the B-Tree). The page (node) size is typically large enough to contain hundreds of elements. So even with a large number of members (millions), the height of a B-Tree is small (2-3 levels). The root page is almost always cached, and location or insertion of elements using a B-Tree requires quite few (1-2) reads/writes of B-Tree pages.

Fortunately, a programmer needn't know all the details of the B-Tree implementation in order to use it (although it is useful to know the underlying nature of the index). In Perst, a simple index is created using the `createIndex` method of the `Storage` class; in fact, this method is used as a factory for different kinds of indexes:

```
Index<MyClass> myIndex =
    db.<MyClass>createIndex(String.class, // key type
                            true); // unique index
```

The first parameter of the `createIndex` method specifies the data type of the key, and the second parameter specifies whether the index is unique or allow duplicates. The `Index` class is derived from the `GenericIndex` class and provides methods for strict match search, range search, prefix search (string keys only) and various iteration methods. Results of a search can be returned as

- a single value (only for unique indexes; if a value is not found, `null` is returned)
- an array of objects matching the search criteria and in the specified order
- an iterator through objects matching the search criteria in a specified direction (ascending or descending)

Because Java versions prior to 5.0 don't support implicit value boxing (transformation of scalar values into objects, for example, `int->java.lang.Integer`), and also because a range-type boundary (inclusive/exclusive) must be specified, in Perst a key is specified using the `Key` class. Overloaded constructors of this class accept all possible types of keys

and optional specifications of boundary type. For string keys (which are used most frequently) it is possible to pass a key value directly without creating a `Key` object instance. If a boundary is not specified (open interval), then a `null` value should be passed.

Java 5.0 version of Perst allows the use of generic versions of indexes. For such indexes, the developer can specify index members' type as a template parameter, in order to avoid extra type casts and to allow the compiler to perform extra type-checking. The code fragments below illustrate all these cases.

Strict match search in unique index:

```
Index<MyPersistentClass> myIndex =
    db.<MyPersistentClass>createIndex(
        int.class, // integer key
        true); // unique index
// Construct a key with an int value 1001 and
// perform a strict match search in the index
MyPersistentClass obj = myIndex.get(new Key(1001));
if (obj != null) { // check if object is found
    // do something
} else {
    // there is no object with that key in the index
}
```

Strict match search in non-unique index:

```
Index<MyPersistentClass> myIndex
    = db.<MyPersistentClass>createIndex(
          String.class, // string key type
          false); // allow duplicates
Key key = new Key("A.B");
ArrayList<MyPersistentClass> list = myIndex.getList(key, key);
for (MyPersistentClass obj : list)
{
    // iterate through selected objects
}
```

Range search:

```
Index<MyPersistentClass> myIndex =
    db.<MyPersistentClass>createIndex(
        int.class, // integer key
        false); // allow duplicates

// Iterate through the records belonging to
// the specified key range
for (MyPersistent obj : myIndex.iterator(
    new Key(100, true), // inclusive low boundary
    new Key(10000, false), // exclusive high boundary
    Index.ASCENT_ORDER)) // ascent order
```

```
{
    // iterate through selected objects
}
```

Iteration though all objects in the index in ascending order:

```
Index<MyPersistentClass> myIndex
    = db.<MyPersistentClass>createIndex(
          String.class, // string key type
          false); // allow duplicates
for (MyPersistentClass obj : myIndex)
{
    obj.doSomething();
}
```

In Perst, index maintenance is the programmer's responsibility. This means that the programmer should insert objects in the proper indexes, and also take care to delete objects from the indexes when needed.

The Index class provides two methods for inserting objects into an index: `put` and `set`. Each has slightly different semantics. The `put` method inserts an object into a unique index only if no object with the same key already exists in that index. Otherwise, it returns false. The `set` method replaces a previously associated object with a new one and returns the previous object, or returns null if there was no such object already in the index.

An object can be deleted from the index using the `remove` method. It requires the programmer to specify the key, and in the case of a non-unique index the object to be removed (in other words, a unique index only requires the key, and not also the object). If there is no such object in the index, then the StorageError(StorageError.KEY_NOT_FOUND) exception is thrown. Please note that removing an object from an index does not remove the object itself from the database. Deleting an object from the storage requires explicitly calling the Deallocate method defined in the Persistent class.

The code fragment below illustrates inserting and removing objects from the index:

```
Index<MyPersistentClass>  myIndex =
    db.<MyPersistentClass> createIndex(
        int.class, // integer key
        false); // allow duplicates
if (!myIndex.put(new Key(1), obj)) {
    // insert object in the index
    reportError("Object with such key already exists");
}
myIndex.remove(new Key(1), obj);
// remove object from the index
```

4.2. Field index

In most cases, a key is stored in one of the object fields. To make insert/remove operations in such cases more convenient, Perst provides a special kind of index, the

`FieldIndex`. With this index, the programmer needn't explicitly specify the key type. Instead, when the index is created, the programmer specifies the name of the indexed field (key) and Perst itself will fetch the value of this field. The Field index is created using the `createFieldIndex` method:

```
FieldIndex<MyPersistentClass> myFieldIndex
   = db.<MyPersistentClass> createFieldIndex(
        MyPersistentClass.class,// indexed class
        "intKey", // name of indexed field
        true); // unique index
```

As with normal indexes, objects can be inserted in a field index using `put` or `set` methods and removed with the `remove` method. But with field indexes it is not necessary to specify a key value, because this value is extracted from the object:

```
myFieldIndex.put(obj);
myFieldIndex.remove(obj);
```

When using JDK 1.5, Perst's field index also implements the `java.util.Collection` interface, which enables use of the `Collection.add` and `Collection.remove` methods.

If an object's key value is updated, the programmer must maintain the index, so the object should first be deleted from the index (with the old value of the key), and then an update performed to reinsert the object into the index with a new key value:

```
myFieldIndex.remove(obj);
obj.intKey = 2;
myFieldIndex.put(obj);
```

4.3. Spatial index

The widespread use of GPS devices (especially in PDAs and mobile phones), and the popularity of services such as Google Maps, significantly increases the demand for applications to work with geographical data. But normal database indexes can't be used to locate an object nearest to the user's current location, or to select a point of interest in the immediate neighborhood.

This requires an index that is specialized for working with multidimensional data. Perst provides such a spatial index based on Guttman's R-Tree algorithm. Perst `Storage` contains two methods for creating spatial indexes: `createSpatialIndex` for a spatial index based on a two-dimensional rectangle with integer coordinates, and `createSpatialIndexR2` for a spatial index based on a two-dimensional rectangle with floating point coordinates.

To work with a spatial index, the programmer needs to specify a *wrapping rectangle*. If an object is represented by point coordinates, then the wrapping rectangle is a degenerated rectangle in which the width and height are zero. For all other geographical objects (lines, polygons, arbitrary shapes), the wrapping rectangle is such that the coordinates of the top left corner are smaller than or equal to the coordinates of any point

of the object, and the coordinates of the bottom right corner are greater than or equal to the coordinates of any point of the object. In other words, a *wrapping rectangle* is the smallest rectangle that fully contains the specified object.

To perform a search for all objects whose distance from the specified point doesn't exceed D, the program performs a search, in the spatial index, of rectangle <x-D,y-D,x+D,y+D> and for all the objects returned, calculates and checks the actual distance (which can be larger than D, because the distance between the center of a square and its corner is D*sqrt(2)). To locate the restaurant that is nearest to the current location, the program performs several iterations: first an initial distance D (let's say 100m) is chosen and the rectangle <x-D,y-D,x+D,y+D> is searched. If there are no restaurants in the specified rectangle, the distance is doubled by searching the rectangle <x-D*2,y-D*2,x+D*2,y+D*2> and so on, until the rectangle contains some restaurants. Then the program calculates the distance for each restaurant, and selects the closest one.
This fragment of code inserts/searches and deletes spatial objects:

```
// create spatial index
SpatialIndex<SpatialObject> index
  = db.<SpatialObject> createSpatialIndex();

// Create object with geographical coordinates
SpatialObject so = new SpatialObject();
// wrapping rectangle
Rectangle r = new Rectangle(top, left, bottom, right);
index.put(r, so); // insert object in spatial index

// get list of objects which wrapping rectangles intersect r
ArrayList<SpatialObject>  sos = index.getList(r);

index.remove(r, so); // remove object from the index
```

4.4. Multifield index

Often an application requires a compound index, or a key consisting of several values—for example, when a person's first and last names both must be considered during a search. To address this, Perst supports both compound and multi-field indexes. A compound index is similar to a normal index, but its key consists of multiple parts:

```
Index<Person> compoundIndex
    = db.<Person>createIndex(
        new Class[] { // key consists of two string values
            String.class, String.class
        },
        true); // index is unique
compoundIndex.put(new Key(new object[] {
                  "Smith", "John" }), person);
```

A multi-field index is similar to a field index, but its key contains several fields:

```
FieldIndex<Person> multifieldIndex
     = db.<Person>createFieldIndex(
```

```
            Person.class, // indexed class
            new String[] { // key consists of two string fields
                "lastName", "firstName"
            },
            true); // index is unique
Person person = new Person("John", "Smith");
multifieldIndex.put(person);
```

To perform a search using a compound or multi-field index, first specify the values of all the key components or the values of the first K components (for a partial key search):

```
// Get person by last and first name
Person person = compoundIndex.get(
    new Key(new object[] { "Smith", "John" }));

// Locate all persons with specified last name
Key lastName = new Key((new object[] { "Smith" });
Iterator<Person> iterator
    = multifieldIndex.iterator(
         lastName, // low boundary(default is inclusive)
         lastName, // high boundary (also inclusive)
         Index.ASCENT_ORDER);
while (iterator.hasNext()) {
    person = iterator.next();
}
```

Please note that searching by first name only is not permitted. A compound index requires that the entire key prefix be specified. When this is impossible, an alternative is the *multidimensional* index, described below.

4.5. Multidimensional index and query-by-example

A typical search form in an application allows the user to specify multiple search criteria. For example, a car can be located by specifying a range for its price, year, mileage, etc. Certainly, an index can be used to pinpoint one of the specified fields (price, for example) and then objects can be filtered to match other criteria. But this is inconvenient for the programmer and inefficient (because it is hard to choose criteria with the highest selectivity, and no such criteria may exist—in which case, the application must inspect a large number of records to select the ones matching the search conditions).

A multidimensional index can be very useful here. Like multi-field and compound indexes, a key in a multidimensional index consists of multiple values. In a multi-field index, the order of the key components is very important: an index on <lastName,firstName> enables searching for persons by specifying both first name and last name or only last name. But it does not allow a search using only first name. A multidimensional index, however, allows any combination of key components.

A multidimensional index described above, the R-Tree, is used for spatial search. Perst supports two-dimensional rectangles; it is also possible to extend the R-Tree algorithm to handle any number of dimensions. But using a spatial index for searches with multiple search criteria is problematic, because the R-Tree requires all dimensions to be specified

and to have the same type (double, for example). If an application must allow users to specify the color of a car (string type) as well price range, the R-Tree is not suitable.

To solve this problem, Perst provides another index, the KD-Tree (K-dimensional tree). There are two ways to define such a multi-dimensional tree:

1. By specifying a multi-dimensional comparator (a special class used to compare different components of a key):

```
static class Stock extends Persistent {
    String symbol;
    float  price;
    int    volume;
};

static class StockComparator
     extends MultidimensionalComparator<Stock>
{
    public int compare(Stock s1, Stock s2, int component) {
        switch (component) {
        case 0: // Stock.symbol
            if (s1.symbol == null && s2.symbol == null) {
                return EQ;
            } else if (s1.symbol == null) {
                return LEFT_UNDEFINED;
            } else if (s2.symbol == null) {
                return RIGHT_UNDEFINED;
            } else {
                int diff = s1.symbol.compareTo(s2.symbol);
                return diff < 0 ? LT : diff == 0 ? EQ : GT;
            }
        case 1: // Stock.price
            return s1.price < s2.price
                ? LT : s1.price == s2.price ? EQ : GT;
        case 2: // Stock.volume
            return s1.volume < s2.volume
                ? LT : s1.volume == s2.volume ? EQ : GT;
        default:
            throw new IllegalArgumentException();
        }
    }

    public int getNumberOfDimensions() {
        return 3;
    }

    public Stock cloneField(Stock src,
                            int component) {
        Stock clone = new Stock();
        switch (component) {
        case 0: // Stock.symbol
            clone.symbol = src.symbol;
            break;
        case 1: // Stock.price
            clone.price = src.price;
            break;
```

```
            case 2: // Stock.volume
               clone.volume = src.volume;
               break;
            default:
               throw new IllegalArgumentException();
            }
            return clone;
        }
    }

    ...
    MultidimensionalIndex<Stock> index
        = db.<Stock>createMultidimensionalIndex
            (new StockComparator());
```

2. Using reflection to generate such a comparator automatically:

```
static class Quote extends Persistent
{
    int    timestamp;
    float  low;
    float  high;
    float  open;
    float  close;
    int    volume;
}

...

MultidimensionalIndex<Quote> index
    = db.<Quote>createMultidimensionalIndex(
        Quote.class, // class of index elements
        new String[] { // list of searchable fields
           "low", "high", "open", "close", "volume"
        },
        false); // do not treat 0 as undefined value
```

A search using a multidimensional index can take the form of a *query-by-example* approach. This approach is very simple: for a search for objects of a class with restricted field values, just create an empty instance of this class and assign the required values to the relevant fields. Consider the following class:

```
class Car {
    String  model;
    String  make;
    String  color;
    int     mileage;
    int     price;
    int     productionYear;
    boolean airCondition;
    boolean automatic;
    boolean navigationSystem;
};
```

Given that class, the following code searches for green Fords:

```
Car pattern = new Car();
pattern.make = "Ford";
pattern.color = "green";
ArrayList<Car> cars = index.queryByExample(pattern);
```

But what if the desired search is for cars priced between $1000 and $2000, with mileage less than 100,000 and including air conditioning? In this case, use two pattern objects, specifying the upper and lower boundaries:

```
Car low = new Car();
// no lower boundary for this field
low.mileage = Integer.MIN_VALUE;
low.price = 1000;
// no lower boundary for this field
low.productionYear = Integer.MIN_VALUE;
low.airCondition = true; // air condition is required

Car high = new Car();
high.mileage = 100000;
high.price = 2000;
// no upper boundary for this field
high.productionYear = Integer.MAX_VALUE;
high.airCondition = true;// air condition is required
high.automatic = true; // full range: [false,true]
high.navigationSystem = true; // full range: [false,true]

ArrayList<Car> cars = index.queryByExample(low, high);
```

Here it is essential to specify boundaries for all scalar and boolean fields (encompassing all potential allowed values, not just the ones used in this query) or to set a minimum/maximum value for this type. If this is inconvenient, the default field value can be excluded from the search conditions. The default value for numeric (integer and floating point) types is 0. The last parameter `treatZeroAsUndefinedValue` of `createMultidimensionalIndex` makes it possible to ignore all fields of numeric type with a 0 value. In our example, mileage can be 0 for a new car, but the boolean fields' ranges still have to be specified.

There are several limitations of the current KD-Tree implementation in Perst (this may change in the future):

1. The KD-Tree now is represented as a binary tree, in which nodes contain left and right pointers, and a pointer to the object. This means that a search using a KD-Tree requires fetching a significantly larger number of nodes than, for example, a search that uses a B-Tree. Also, the search key is not located in a node itself (as in the case of B-Trees), but is fetched from a referenced object. As a result, the current implementation is most suitable

for in-memory databases (databases which reside entirely in main memory).

2. Currently, no balancing of the KD-Tree is performed. Inserting objects in a "bad" order can cause degeneration of the tree, resulting in a longer search time.
3. The KD-Tree is never truncated. When some object is removed from the tree and it is referenced from a non-leaf node, it is replaced with a stub.

4.6. Specialized collections: Patricia Trie, bitmap, thick and random access indexes

Perst provides other useful collection classes, described in the Perst API documentation. Briefly, here are a few key features of these collections:

Patricia Trie

The Patricia Trie is the most efficient data structure for prefix searches—for example, in a table containing information about telephone operators and their numerical prefixes, the Patricia Trie can efficiently locate the proper operator to handle a received call. This ability is applicable to IP addresses and masks, and hence to algorithms used in IP routers, filters, and other telecommunications and network communications applications.

Bit index

This is another data structure for multidimensional search, and is most efficient when a class has a large set of characteristics with a small range of possible values.

Thick index

This index, based on the B-Tree, is optimized for keys with large numbers of duplicates. Removing such objects from a standard B-Tree is very inefficient because it requires sequential searching through all elements with the same key value. The Thick index adds some extra overhead in terms of code size and memory consumed, but the complexity of the remove operation remains the same (O(log(N)) even if all N objects contain the same key value.

Random access index

When a data set is presented visually in a user interface (UI), an application frequently must locate elements by their position (for example, to perform navigation and scrolling in UI form). The standard B-Tree can't efficiently locate elements by position (because sequential traversal starting from the first element is required). The Random access index solves this problem while adding only minimal extra code footprint and memory overhead.

Note that Perst collections typically are, themselves, normal persistent objects. So the developer can easily create new collections that best fit an application's requirements.

5. Transaction model

5.1. Shadow objects and log-less transactions

Although the main goal of object-oriented database systems is to provide transparent persistence—to eliminate any difference between working with persistent and transient objects—some aspects of persistent data cannot and should not be hidden from the programmer. One of them is transaction control. Transactions are a fundamental mechanism of database systems, and serve two main goals: enforcing database consistency, and providing concurrent access to the database by multiple clients. The transaction body is the atomic sequence of logically related operations that should all be accepted together, or rejected together.

Perst uses a shadow object transaction mechanism. When the application modifies an object, this process does not rewrite the object directly; rather, a copy of the object is created and updated. During transaction commit, originals of the objects are replaced with their updated copies. This is achieved atomically, switching the database from one consistent state to another.

A shadow object transaction mechanism has several advantages:

1. The need for a transaction log file is eliminated, allowing all data to be stored in single database file
2. Fast recovery
3. Dynamic re-clustering of objects that are accessed together

The main disadvantage of the approach is its difficulty supporting multiple concurrent transactions (the transaction log approach allows many entries in the log, each corresponding to a different transaction, but shadow objects permit only two states for the database: original and new). However, Perst offers a work-around for this limitation, described below.

Perst's guiding principle is to provide programmers full control over interaction with the database, without introducing extra overhead or limitations. In keeping with this rule, concurrency control and providing transaction isolation levels are the responsibility of the programmer.

5.2. Transaction modes

Perst supports the following basic modes of synchronization:

Synchronized access to the database

All access to the database is performed from one thread. No synchronization is needed at the database level.

Cooperative transactions

- Multiple threads can share a transaction, and all of them must "see" the others' changes. The programmer should use locking to avoid race conditions. Tools to accomplish this include the Java/C# internal synchronization mechanisms as well as the Perst locking mechanism (see section 5.3, *Object locking*).
- Committing a transaction in this mode stores to disk all changes made by the threads, and a rollback of the transaction undoes the work of all threads.

- Cooperative transactions should be used with special care, because different threads' lack of isolation can cause synchronization problems that are hard to detect, reproduce and eliminate.
- In this mode, the programmer only uses two methods from the `Storage` class: `commit` and `rollback`. There is no need to explicitly start a transaction. A new transaction is implicitly started when the old one is committed.

Exclusive per-thread transactions

- This mode, in which each thread accesses the database exclusively, is the safest, because unintended interaction between threads is impossible. No locking is needed. But it represents the worst case for concurrency, since at any time, only one transaction can be executed (even if it is a read-only transaction). So this should be used only for applications with few concurrent activities needing access to the persistent data.
- In this mode, exclusive transactions should be started using `Storage.beginThreadTransaction(Storage.EXCLUSIVE_TRANSACTION)` and finished either by `Storage.endThreadTransaction()` or by `Storage.rollbackThreadTransaction()`:

```
db.beginThreadTransaction(Storage.EXCLUSIVE_TRANSACTION);
try {
    // do something
    // commit changes in case of normal completion
    db.endThreadTransaction();
} catch (Exception x) {
    // rollback transaction in case of exception
    db.rollbackThreadTransaction();
}
```

Serializable per-thread transactions

The notion of *serializable transactions* means that transactions executing in parallel do not overlap or affect each others' outcomes. In other words, it means that each transaction can work as if it is the only transaction accessing the database. However, this "serializability" of the transaction is not enforced by the Perst database engine—it is the programmer's responsibility to set proper locks. One database system theorem says that to achieve serializability, it is sufficient to apply locks to all accessed objects until the end of the transaction. Further, this thinking holds that for an object to be accessed in read-only mode, it is enough to set a shared lock; updating an object requires an exclusive lock. The Perst locking mechanism should be used by the application to lock objects. With serializable transactions, locks are automatically released during transaction commit, so there is no need to unlock objects explicitly.

Serializable transaction should be started using `Storage.beginThreadTransaction(Storage.SERIALIZABLE_TRANSACTION)` and finished either by `Storage.endThreadTransaction()` or `Storage.rollbackThreadTransaction()`:

```
class MyClass extends PersistentResource
```

```
// persistence-capable class with resource control
{
    void someReadOnlyMethod() {
        // this method only reads the state of the object
        sharedLock();
        // prevents modification of the object by
        // other transactions
        ...
    }

    void someUpdateMethod() {
        // method updates the state of the object
        exclusiveLock(); // exclusive access to the object
        ...
    }
}

class Activity
{
    MyClass obj1;
    MyClass obj2;

    void run() {
        Storage db = obj1.getStorage();
        // start serializable transaction
        db.beginThreadTransaction(
            Storage.SERIALIZABLE_TRANSACTION);
        try {
            obj1.someReadOnlyMethod();
            obj2.someUpdateMethod();
            // commit changes in case of normal completion
            db.endThreadTransaction();
        } catch (Exception x) {
            // rollback transaction in case of exception
            db.rollbackThreadTransaction();
        }
    }
}
```

As explained above, Perst implements transactions using a “shadow objects” mechanism and is not able to handle more than one concurrent transaction. Serializable per-thread transactions can be executed concurrently because Perst holds all modified objects in memory, so the database file is not changed until the transaction is committed. Therefore, a transaction rollback just means that the database “forgets” about all modified objects. And when the `Storage.endThreadTransaction()` method is executed, then for a short time, Perst sets an exclusive lock and performs a normal transaction commit: it saves modified versions of objects in the storage, and atomically switches the storage to the new consistent state.

The obvious disadvantage of this approach is that the transaction size is limited by the amount of memory available for the application. Overly large transactions can cause memory overflow. Another potential complication arises from Perst’s internal use of B-Trees to implement various indexes. Unlike all other persistent objects, a B-Tree interacts

directly with the page pool. This design provides better performance, since B-Tree pages are fetched and stored directly in the page pool. But it interferes with serializable transactions that are based on pinning objects in memory. So, when using serializable transactions, an application should take advantage of Perst's alternative B-Tree implementation, which stores B-Tree pages as persistent objects that are pinned in memory like all other objects. Switching to this alternative B-Tree implementation requires setting the "perst.alternative.btree" property before opening the storage:

```
// alternative B-Tree implementation is needed
// for serializable transactions
db.setProperty("perst.alternative.btree", Boolean.TRUE);
db.open("testdb.dbs", pagePoolSize); // open storage
```

When the database is opened, Perst checks if it was normally closed (that the `Storage.close()` method was called before application termination). If it wasn't, Perst automatically performs database recovery, restoring a consistent database state corresponding to the last committed transaction. It is important to note that the `Storage.close()` method implicitly triggers a commit and so commits all uncommitted changes. Therefore, if the application is terminated abnormally (for example, because of some critical exception), the developer must be cautious about trying to close the storage, since this action can cause the inconsistent state to be committed, preventing the recovery that would normally occur when the database is opened:

```
public class Application
{
    public static void main(String[] args)
    {
        // get instance of the storage
        Storage db =
            StorageFactory.getInstance().createStorage();
        // open the database
        db.open("test.dbs", pagePoolSize);
        try {
            // do something with the database
            db.close();
        } catch (Throwable x) {
            System.err.println("Catch " + x);
            // Do not close the database,
            // let recovery mechanism do it at next opening
        }
    }
}
```

5.3. Object locking

As its locking mechanism, Perst provides the `PersistentResource` class, derived from Persistent, which is assumed to be the base class for all classes requiring synchronization. All Perst collections are derived from the `PersistentResource` class in order to use it as a synchronization root. The `PersistentResource` class provides methods for setting shared and exclusive locks and for unlocking the object. Many threads can concurrently set shared locks, but only one thread can set an exclusive lock. Locks can be released

explicitly by calling the unlock method, but in the case of serializable transactions, this should not be done, because locks are automatically released at the end of the transaction. Two main challenges of locking are as follows:

- The possibility of forgetting to lock some object and thereby causing a race condition.
- Deadlock, if two or more threads are locking the same objects in different order, or if both are trying to upgrade their shared lock to exclusive for the same object. A deadlock can stop both threads' processing.

When using the standard Java synchronization primitives, the developer must choose the proper locking policy to prevent race conditions and deadlock. Perst offers a separate "Continuous" package which provides object versioning, optimistic locking, and full text search. With the addition of this technology, all required synchronization is handled by Perst, preventing deadlocks and race condition. The Continuous package is described in a separate document.

5.4. Multi-client mode

The Perst embedded database is meant to be accessed by one client (process). But some applications require multi-client access, including access from remote clients in a network. Perst for Java supports multi-client database access based on the standard file system locking mechanism. In this mode, transactions to modify the database are exclusive (only one transaction can concurrently update the database), while read-only transactions can run in parallel. Using locks is not needed in read-only mode.

To enable multi-client mode, set the "perst.multiclient.support" property for the storage accordingly, before the database is opened. All access to the database should be performed within a transaction body. Transactions are started by the `Storage.beginThreadTranaction(mode)` method where the mode is either `Storage.READ_WRITE_TRANSACTION` or `Storage.READ_ONLY_TRANSACTION`:

```
// enable multiclient access
db.setProperty("perst.multiclient.support", Boolean.TRUE);
db.open("testdb.dbs", pagePoolSize); // open storage

// start read-only transaction
db.beginThreadTransaction(Storage.READ_ONLY_TRANSACTION);
try {
    // do something
    db.endThreadTransaction();
} catch (Exception x) {
    db.rollbackThreadTransaction();
}

// start read-write transaction
db.beginThreadTransaction(Storage.READ_WRITE_TRANSACTION);
try {
    // do something
    db.endThreadTransaction();
} catch (Exception x) {
```

```
            db.rollbackThreadTransaction();
        }
```

6. Relational database wrapper

6.1. Emulating tables

Perst's purpose is to let the programmer work with persistent objects in almost the same way as with normal transient objects. The "pros" of this approach are elimination of overhead, and greater efficiency for tasks such as maintaining class extents (the set of all instances of the class), object locking, comparing old and new object instances to detect updated fields, updating indexes, parsing queries, optimization and processing. But the major "con" is extra work for the programmer, who must maintain indexes, manage locking, write code to implement queries, etc.

This work can be avoided by Perst's `Database` class, which provides a more convenient API that is similar to those used in relational database systems (this is not supported in Perst Lite, the version of Perst for Java ME).

The `Database` class provides the following functionality:

1. Maintains class extents (tables) that allow iterating through all instances of the particular class. The programmer does not need to create the root object.
2. Maintains indexes: these indexes are either explicitly created by the programmer, either implictly created by the `Database` class based on relevant field annotations (Java 5.0), either automatically created on demand by the `Database` class when *auto-indices* mode is enabled (`Database.enableAutoIndices` method). `Database` automatically includes an object in all indexes when the object is inserted in the database, and removes it from indexes when the object itself is removed. But object updates must be explicitly handled by the programmer: an object should be removed from an index before it is updated, and reinserted in the index after it is updated.
3. Provides serializable transactions using table-level locking.
4. Provides a query language, JSQL, which is Perst's object-oriented subset of SQL. A JSQL query returns a set of objects, not tuples. It doesn't support joins, nested selects, grouping or aggregate functions.

The `Database` class can be termed a relational database wrapper because it provides associations between an RDBMS table and a Java class, as well as between row and object instances. In the Java 1.5 version of Perst, it is possible to mark fields of the class for which indexes should be created using the `Indexable` annotation:

```
    static class Record extends Persistent {
        @Indexable(unique=true, caseInsensitive=true)
        String key;
```

```
    String value;
}
```

When using this wrapper, to open the database, first open the storage and pass it to the `Database` class constructor:

```
Storage storage = StorageFactory.getInstance().createStorage();
// create in-memory storage
storage.open(new NullFile(), Storage.INFINITE_PAGE_POOL);
Database db = new Database(storage,  // opened storage
                           true);    // allow multithreaded
                                     // access
...
db.close();
```

To add a record to the database, create an instance of a persistence-capable class (derived from `Persistent` or another persistence-capable class) and invoke the `Database.addRecord` method. This method will automatically insert the record in all indexes. All database access should be performed within a transaction body that is enclosed by `Database.beginTransaction/Database.commitTransaction` methods:

```
db.beginTransaction();
try {
    Record rec = new Record();
    db.addRecord(rec);
    db.commitTransaction();
} catch (Exception x) {
    db.rollbackTransaction();
}
```

The application can iterate though all instances of the class using the `getRecords` method:

```
for (Record rec : db.<Record>getRecords(Record.class)) {
    rec.dump();
}
```

It is possible to define a JSQL query:

```
for (Record rec : db.<Record>select(Record.class,
     "key like 'ABC%'"))
{
    System.out.println(rec);
}
```

If a query must be executed multiple times, the application can *prepare* the query in order to reduce query parsing overhead for each execution. A prepared query can contain positioned parameters (specified by the '?' character). Before execution, values should be assigned to parameters using the `Query.setParameter`, `Query.setIntParameter`, `Query.setRealParameter` or `Query.setBoolParameter` methods. The parameter index is 1-based:

```
        Query<Record> query = db.<Record>prepare(Record.class,
"key=?");
        // '?' is parameter placeholder
        for (int i = 1; i < 1000; i++) {
            query.setIntParameter(1, i); // bind parameter
            Record rec = query.execute().next();
            rec.doSomething();
        }
```

To update or delete selected records, perform a "select for update" and pass `true` as the second optional parameter `forUpdate` of the `select` method. Perst then sets an exclusive lock on the table, to prevent other concurrent transactions from modifying it. When updating a record, remember to call the `Persistent.modify()` method to mark the record as modified. If a key field of the record is changed, the object must first be deleted from the corresponding index, and then inserted again using an update:

```
        Iterator<Record> i = db.<Record>select(
            Record.class, // target table
            "key='1'", // selection predicate
            true); // select for update
        if (i.hasNext()) { // if record is found
            Record rec = i.next();
            // exclude if from index before update
            db.excludeFromIndex(rec, "key");
            rec.key = "2";  // update record
            rec.modify();  // mark record as modified
            db.includeInIndex(rec, "key");  // reinsert in index
        }
```

To delete a record from the database, call the `deleteRecord` method:

```
        Record rec = db.<Record>select(Record.class, "key='2'",
                                       true).next();
        db.deleteRecord(rec);
```

6.2. JSQL query language

JQSL is Perst's object-oriented subset of SQL. It provides almost the same syntax as standard SQL. The main difference is that a JSQL query returns a set of objects, not tuples. In addition, JSQL doesn't support joins, nested selects, grouping and aggregate functions. When defining a JSQL query, it is not necessary to specify a "select ... from ... where" clause because, as mentioned above, JSQL always selects objects, and the relevant table is specified as a parameter of the select method.

All Perst collections accept a JSQL query as a filter. Please note that in this case, Perst traverses all collection members (so it is a sequential search through the collection).

The Database class provides a simple optimizer for JSQL queries: it is able to use existing indexes if the query predicate uses indexed fields. But indexes can be used in this way only within a single table – if the search predicate also contains some condition for a field in a referenced table, then this query will require a sequential scan.

The following rules, in BNF-like notation, specify the grammar of JSQL query language search predicates:

Grammar conventions

Example	Meaning
expression	non-terminals
not	Terminals
|	disjoint alternatives
(**not**)	optional part
{**1**..**9**}	repeat zero or more times

```
select-condition ::= ( expression ) ( traverse ) ( order )
expression ::= disjunction
disjunction ::= conjunction
        | conjunction or disjunction
conjunction ::= comparison
        | comparison and conjunction
comparison ::= operand = operand
        | operand != operand
        | operand <> operand
        | operand < operand
        | operand <= operand
        | operand > operand
        | operand >= operand
        | operand (not) like operand
        | operand (not) like operand escape string
        | operand (not) in operand
        | operand (not) in expressions-list
        | operand (not) between operand and operand
       | operand is (not) null
operand ::= addition
additions ::= multiplication
        | addition +  multiplication
        | addition || multiplication
        | addition -  multiplication
multiplication ::= power
        | multiplication * power
        | multiplication / power
power ::= term
        | term ^ power
term ::= identifier | number | string
        | true | false | null
       | current
       | ( expression )
        | not comparison
       | - term
       | term [ expression ]
       | identifier . term
       | function term
        | count (*)
        | contains array-field (with expression) (group by identifier
having expression)
        | exists identifier : term
function ::= abs | length | lower | upper
```

```
        | integer | real | string |
        | sin | cos | tan | asin | acos |
        | atan | log | exp | ceil | floor
        | sum | avg | min | max
string ::= ' { { any-character-except-quote } ('') } '
expressions-list ::= ( expression { , expression } )
order ::= order by sort-list
sort-list ::= field-order { , field-order }
field-order ::= field (asc | desc)
field ::= identifier { . identifier }
fields-list ::=  field { , field }
user-function ::= identifier
```

Identifiers are case sensitive, begin with a..z, A..Z, '_' or '$' character, contain only a-z, A..Z, 0..9 '_' or '$' characters, and do not duplicate any SQL reserved words.

List of reserved words

abs	acos	and	asc	asin
atan	avg	between	by	contains
cos	ceil	count	current	desc
escape	exists	exp	false	floor
group	having	in	integer	is
length	like	log	lower	max
min	not	null	or	real
sin	string	sum	tan	true
upper	with			

More information about JSQL functions can be found in the Perst manual.

Below are examples of JSQL queries:

```
db.select(Person.class, "age > 30 and salary < 100000");
db.select(Detail.class, "color='grey' order by price");
db.select(Order.class, "detail.delivery between '01/01/2008'" +
                       "and '02/01/2008'");
db.select(Book.class, "title like '%DBMS%'");
db.select(Company.class,
         "exists i: (contract[i].company.location = 'US')");
```

7. Advanced topics

7.1. Schema evolution

With Perst, the lifetime of persistent objects exceeds the life-time of an application session. So what happens if application code is changed between sessions—for example, by adding new fields to the class or renaming/removing fields? The database engine must

convert stored persistent objects to the new format. This procedure is called *schema evolution*.

Perst performs automatic schema evolution. This allows the developer to stop worrying about changing the database format. Perst follows the lazy schema evolution strategy, meaning that it doesn't try to convert all objects to the new format immediately, when the database is opened. Instead, an object is converted to the new format when it is loaded. But even then, it is not immediately stored to disk in the new format. This occurs only if the application updates it and commits the transaction. Then the object is stored in the new format and evolution is completed for this object instance. But if an object is not accessed or modified, it remains in the database in the old format. Thus, a storage may contain different generations of objects.

Perst's schema evolution mechanism is based on name-matching, so the database will not be able to handle modifications correctly if a field or class is renamed. More precisely, Perst does not allow the following code changes:

1. Renaming a class
2. Renaming a field
3. Moving a field to a base or derived class
4. Changing class hierarchy
5. Incompatible changes of field type (for example int->String). Perst can convert field types only using the explicit conversion operator in Java

When an application requires one of these modifications, Perst's XML import/export facility must be used to convert the database to the new format.

7.2. Database backup and compaction

Perst aims to allocate objects sequentially and reduce database storage fragmentation. But after intensive object allocation and de-allocation, fragmentation can still occur. The developer can compact the storage using the `Storage.backup(java.io.OutputStream out)` method. Storage backup doesn't require closing the application, but it needs exclusive access to the database (no transactions are allowed while backup is in progress).

Because the Perst database exists in a single file, restoring from backup is very simple: just copy the backup file in place of the original database file. In addition to its usefulness in compaction, the backup function can also be used to restore a database in case of a failure not handled by the transaction recovery mechanism (for example, hard drive corruption).

This code fragment illustrates database backup:

```
        Storage db = StorageFactory.getInstance().createStorage();
        db.open("test.dbs", pagePoolSize); // Open the database
        try {
            // open output stream for backup
            OutputStream out = new FileOutputStream("test.bck");
//perform database backup to the specified stream
```

```
            db.backup(out);
// backup doesn't close the stream, it should be done here
            out.close();
        } catch (IOException x) {
            System.err.println("Backup failed: " + x);
        }
        db.close();
```

7.3. XML import/export

Because Perst is an object-oriented database, its data format can be quite complex. So it is difficult to implement export utilities to popular data formats (such as DBase, Excel CSV, etc.). This is why Perst exports only to the XML format. XML is now considered one of the most universal data exchange formats. A wide range of tools exists for processing XML data; it can even be inspected in a standard Web browser.

Export to the XML is performed using the `Storage.exportXML(java.io.Writer writer)` method, and import is done with the `importXML(java.io.Reader reader)` method. The application performing XML import to a Perst database should include the definition of the corresponding classes (obviously, Perst cannot generate a class file from XML data). Please note that Perst doesn't generate a DTD description of the database scheme.

The following code illustrates XML-based export and import:

```
        Storage db = StorageFactory.getInstance().createStorage();

        db.open("test1.dbs", pagePoolSize); // Open the database
        try {
            Writer writer
                = new BufferedWriter(new FileWriter("test.xml"));
            // export the whole database to the specified writer in XML
            // format
            db.exportXML(writer);
            // exportXML doesn't close the stream, close it here
            writer.close();
        } catch (IOException x) {
            System.err.println("Export failed: " + x);
        }
        db.close();

        db.open("test2.dbs", pagePoolSize);

        try {
            Reader reader
                = new BufferedReader(new FileReader("test.xml"));
            db.importXML(reader); // import data from XML stream
// importXML doesn't close the stream, close it here
            reader.close();
        } catch (IOException x) {
            System.err.println("Import failed: " + x);
        }
        db.close();
```

7.4. Database replication

Database replication means maintaining multiple database copies (replicas) at different network nodes. Replication is used for two main purposes:

1. Fault tolerance
2. Load balancing

If several copies of the database exist, then if one node fails, access can still be provided from other nodes (fault tolerance). Moreover, if one server is unable to serve all client requests, these requests can be redirected to other nodes (load balancing). Replication increases system scalability: if the number of clients increases and existing nodes cannot provide the desired throughput and response time, more nodes can be added, existing clients can be served, and the number of clients can even increase further.

Perst supports master-slave replication: there can only be one master node on which the application updates the database, and one or more slave nodes that receive updates from the master and can execute read-only database requests. Replication is performed at the page level: when a transaction is committed, Perst sends the updated pages to all replicas. Within this architecture, transactions can be implemented asynchronously (by a separate thread, without waiting for acknowledgement from the replica) or synchronously (in which the master waits for acknowledgement from the replica before committing the transaction). To toggle between asynchronous and synchronous replication, specify the "perst.replication.ack" property.

Perst provides yet another replication option: replication can be static (when the number of replicas is fixed at the time the master database is opened) and dynamic, in which new replicas can be connected to the master node at any time. The main difference between these two modes is the time at which the replica is connected to the master - in static mode, all replicas are assumed to be in identical states, so the master only broadcasts modified pages to the replicas. In dynamic mode, a newly-attached replica is assumed to be empty or out of sync. In this case, Perst synchronizes the database state between the master and the new replica node, sending the full content of the database to this replica. Synchronization is performed by a separate thread, so it should not block the master.

Please note that Perst does not itself detect node failure, choose a new master node (failover) or perform recovery of crashed nodes. These actions are the responsibility of the application itself.

Below is an example of the database replication process:

Master side:

```
        ReplicationMasterStorage db =
           StorageFactory.getInstance().createReplicationMasterStorage(
               -1,
// port at which master will accept connections of new
```

```
// replicas,-1 means that connections of new replicas are
// not supported
                new String[]{"localhost:" + port},
// list of slave node addresses
                async ? asyncBufSize : 0); // size of asynchronous
buffer

        // Disable synchronous flushing of disk buffers because
        // in case of fault database can be recovered from slave node
        db.setProperty("perst.file.noflush", Boolean.TRUE);
        // set replication mode
        db.setProperty("perst.replication.ack", Boolean.valueOf(ack));
        db.open("master.dbs", pagePoolSize); // open master storage

        // ... Work with the database

        db.close(); // close master storage
```

Slave node:

```
        // Create replica which accepts master connection at the
        // specified port
        ReplicationSlaveStorage db =

StorageFactory.getInstance().createReplicationSlaveStorage(port);
        // Disable synchronous flushing of disk buffers because
        // in case of fault database can be recovered from slave node
        db.setProperty("perst.file.noflush", Boolean.TRUE);
        // set replication mode
        db.setProperty("perst.replication.ack", Boolean.valueOf(ack));
        db.open("slave.dbs", pagePoolSize); // open slave storage

// Slave node receives modifications from master in separate
// thread. Concurrently it can execute its own read-only
// transactions. But to perform some processing at slave node
// only when some data is changed (transaction is committed by
// master node)then the
// ReplicationSlaveStorage.waitForModifications() method can be
// used to wait for update of the database.
        while (db.isConnected()) { // while master is alive
            // Wait until master commits new transaction
            db.waitForModification();
            // Start special read-only transaction at replica
            db.beginThreadTransaction(
                Storage.REPLICATION_SLAVE_TRANSACTION);
            ... // Do some processing of the database
            db.endThreadTransaction();
        }
        db.close(); // close slave storage
```

8. Perst Open Source License Agreement

Perst is open source: you can redistribute it and/or modify it under the terms of version 3

(or earlier versions) of the GNU General Public License as published by the Free Software Foundation.

If you are unable to comply with the GPL, a commercial license for this software may be purchased from McObject LLC.