FAST OBJECTS™

Performance with FastObjects™ for JDO

A FastObjects White Paper

Table of Contents

1 Introduction

There are many ways in a Java application to store data need to live longer than the actual execution of the program. The Java language even provides a mechanism, called serialization, that allows you to write Java object networks to disk. Although this is an easy way to make data persistent, realistically it can only be used for simple object networks. Furthermore, essential features such as transactions, recovery, and reorganization utilities are missing. A database is needed for this.

The original idea for developing object-oriented database systems was to overcome the "impedance mismatch" that results from the need to map your programming data structure to a different data structure that can be handled by a (relational) database system. This need for mapping from one data structure (objects) to another (tables) not only costs time during development of your application, it limits your data structure modeling possibilities. It also slows down performance when using complex data during runtime of the application.

With object-oriented databases data is stored using the same structure as in your application. One data structure means you don't have to worry about how the data will be stored or how to retrieve your objects back from storage. If you store an object in a FastObjects database and read it back, the object behaves as though it had never been stored. The object you read from the database has the same identity, encapsulation, inheritance structure, polymorphism, and references as the original object. So you can model your data structure to perfectly fit the needs of your application. None of the information regarding the relationship between data is lost when storing data to database. And your objects need not be expensively reconstructed when retrieving data back from database.

2 General Facts to Keep in Mind

When developing applications with FastObjects, you are completely freed from thinking about how your object data is stored and retrieved. Nevertheless, you should always be aware that your data is being stored to and retrieved from a database. Naturally, FastObjects databases reside on secondary storage units (mostly fixed-disk storage). This means that, just like for any other database system, access speed to a FastObjects database is limited by the access speed to the secondary storage provided by your hardware and your operating system. FastObjects is designed to minimize this limitation and provides a great range of sophisticated features to speed up data access in order to accommodate the needs of your specific application.

To enable FastObjects to fully take advantage of its potential, you should know about these features and keep them in mind when designing your application. There is no "silver bullet" that will solve your every problem when working with FastObjects. All of the features and tricks we present in this paper can result in substantial improvements in performance *in specific access scenarios*. However, under certain circumstances these tuning tricks can actually slow down access speed. In this paper we will describe these features and their positive effects but will also show the side effects that can occur. You must determine whether the use of a

specific feature will speed up your own application or whether it actually slows your application down.

We have identified a few common problems that can significantly restrict the potential of FastObjects:

- Using an inappropriate (not object-oriented) database schema
- Performing unnecessary read and write operations
- Performing too many client/server calls
- Keeping unnecessary objects in memory
- Using transactions that are too large or too long lasting
- Using an inappropriate access method
- Building and maintaining too many indexes

These points will be discussed in the following sections.

2.1 Using an Inappropriate (Not Object-Oriented) Database Schema

An object database is a database that fully supports the object-oriented programming model. Like an object-oriented programming language, an object database is designed to express the relationships among data. And, like a conventional database, an object database is designed to manage large amounts of persistent data.

FastObjects is a true object database. It only makes sense to model your object data as a true object-oriented data-model. Much of FastObjects' power comes from the ability to represent complex object networks in the database on a one-to-one basis. When trying to simplify complex interrelationships between objects by modeling a less complex database schema, much of the benefits of using FastObjects is lost. In such cases, you must reconstruct the complexity of the object network in your application. And that means more effort in work and time for development and loss of application speed resulting from having to process the reconstruction. Simplifying the data-model can have the negative consequence of making many of the FastObjects features inappropriate or unavailable to you. FastObjects must maintain certain data constructs and perform background administrative tasks and this, of course, consumes some amount of memory and time. If your design does not allow you to make use of these features these administration efforts are unproductive.

Relational databases do not provide those features simply because they do not have the power to make any use of them. This can lead to the impression that an object database is slower than a relational database– that is, when you simply try to transfer the data structure you developed for a relational database to FastObjects. From a technical point of view, this is no problem. But to consider working with FastObjects as a simple transfer is a false beginning and will not allow you to fully embrace FastObjects. Optimized work with FastObjects begins with, and highly depends on, developing an optimal database schema, one that represents your object model in the same way you will later work with the objects in your application.

2.2 Performing Unnecessary Read and Write Operations

2.2.1 Writing Objects

As pointed out earlier, access speed to a FastObjects database is limited by your hardware and operating system capabilities to access secondary storage. When you write an object, you are making a secondary storage access. In fact, only objects that were newly created or objects that were modified must actually be written. The FastObjects JDO Binding automatically identifies the modified objects in your object network and makes sure that only these objects are written to database. This is one of the major advantages of the FastObjects JDO Binding because you as the application designer need not be concerned about database tasks.

2.2.2 Reading Objects

Reading objects from a FastObjects database does not necessarily mean that the entire object data content is always read from secondary storage. The FastObjects JDO binding provides an automatic transaction cache to cache objects that were read into memory. Its main task is to provide object identity, that is to ensure that every object has just one presentation in memory.

However, this cache only works for one specific client at once, because it is located on the client side. If your application has a client/server scenario where several clients are connecting to one or more servers, this cache management cannot be used, because data integrity is not ensured across the different client caches. For such scenarios FastObjects provides a sophisticated cache management system to keep the caches of the involved clients in synchronization. This feature is called Active Java Cache. It will be discussed in depth later in this article.

2.2.3 Performing Too Many Client/Server Calls

One of the most common uses of a database is the sharing of data among concurrent users. For this purpose, client/server scenarios are set up where one database server provides the requested data to many different applications. These applications may be running on the same or on different computers. The freedom of storing the data in one place and accessing it from anywhere is obtained with some administration effort. That naturally means a loss in performance compared to direct database access. This is especially true when applications operate via networks. The number of client/server calls can, and should, be minimized by reading and writing groups of objects rather then to read and write every object individually. FastObjects provides a range of features to bundle server calls. They will be discussed later in this article.

2.2.4 Keeping Unnecessary Objects in Memory

Every object loaded into memory consumes a specific amount of RAM space. Too many objects can cause the system to run low on RAM space and to begin swapping data to secondary storage. This can significantly slow the application. Strict memory management is therefore essential for high performance application execution. The Java programming language provides a garbage collection mechanism to automatically remove objects from memory that are no longer referenced. However you should

pay attention that the garbage collector can do its work by immediately releasing references to objects that are no longer needed by your application. You can help the garbage collector by explicitly marking objects that are not needed, allowing FastObjects to remove them from the cache. This will also be discussed.

2.3 Using Transactions that Are Too Large or Too Long Lasting

Transactions are series of operations that succeed or fail as a unit. Transactions allow you to tell FastObjects to make changes to a database tentatively and to save or abandon any series of changes as a whole. In this sense, transactions are responsible for data integrity. With the FastObjects JDO Binding, you always need to use transactions for accessing objects from the database.

Large transactions are transactions that contain many objects. They will likely require much memory, thus slowing down the application as pointed out in the previous section. The memory consumption during the lifetime of the transaction, and especially during commit, increases with the number of modified objects.

Long lasting transactions are transactions with a long duration period. They can become a performance problem when they are holding locks to objects that other transactions try to write. As long as one or more transactions hold read locks to an object, no other transaction is allowed to modify this object. And maybe even worse, as long as the long lasting transaction holds a write lock to an object no other transaction is allowed to read it.

2.4 Using an Inappropriate Access Method

FastObjects provides many techniques for accessing objects. In principle, two types of access methods can be defined: relationship-based and value-based access. As FastObjects is an object database, following object relationships (i.e., references) will always provide the best performance. Value-based access should generally only be used to retrieve a “root object” or some other entry point to an object network and following relationships should then be used to access related objects. These access methods are discussed in the following sections.

2.4.1 Using Queries Instead of Alternative Access Methods

As an object database, FastObjects holds the complete description of your classes and objects, their data and their relationships. Therefore, the natural and fastest way to access a specific object is by direct navigation through the object network based on the relationships between the objects.

A value-based access method, in contrast, does not make any use of the modeled relationships. With value-based access, one or more specific objects will be retrieved based on whether the values of its data-members meet a certain condition. Relational database systems lack the capability for direct navigation. In such systems all data must be accessed using the value-based method. For many application designers who are not experienced in using object databases, this seems to be the natural way.

So, often unnecessary queries are used to access data. Access based on direct navigation is simply overlooked.

There are situations, even when using an object database, when objects must be accessed on the basis of the values of their data-members. A query is just one technique used for returning all occurrences of objects satisfying the query specification. FastObjects provides a range of features for value-based access of objects. These features include finding objects by their object ID or by key values, filtered extents for specific classes and executing queries. Finding objects by key values requires that *indexes* be defined. Filtering extents and queries do not necessarily need a defined index but can make use of one. In many cases, the find and filter methods provide much better performance than using queries.

2.4.2 Defining Too Little or Too Many Indexes

When objects have to be retrieved by their member values, using indexes can be much faster than working without them. Finding objects by key values and filtering extents do not even work without a defined index. Queries can also be faster when using indexes. But a significant performance improvement of the application as a whole is not necessarily guaranteed. Every change in the value of an indexed data-member of an object, every creation of a new object which an indexed data-member and every deletion of an indexed object causes an update of the index. This is necessary to ensure that the index always contains the appropriate objects at any given time. Otherwise, searches or queries based on the index cannot be insured to return all valid objects. Index updates are carried out automatically by FastObjects. Although automatic, updates are not instantaneous and take time.

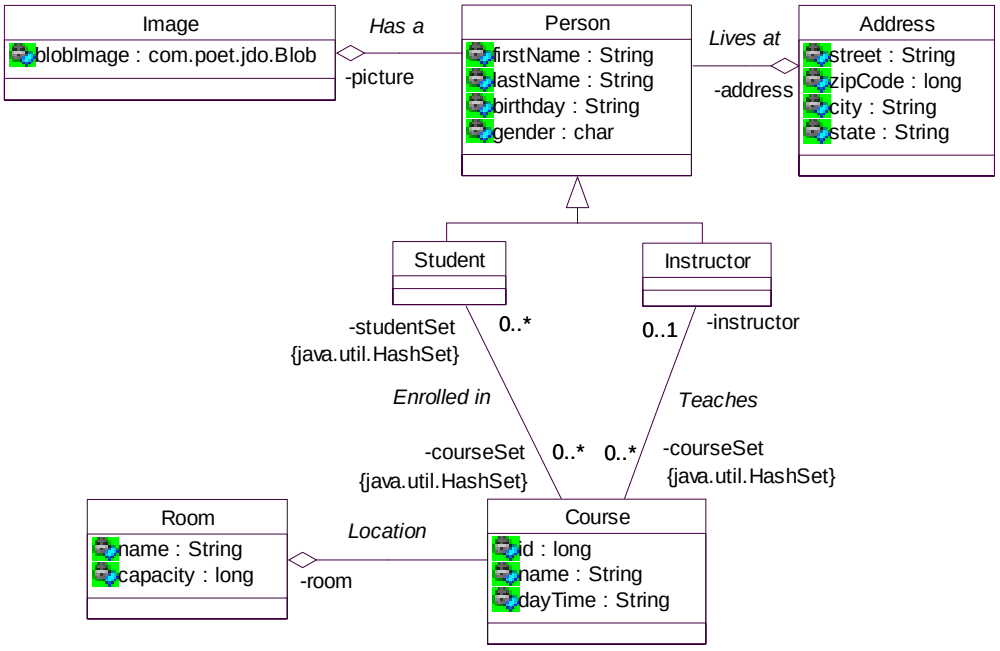
You should think carefully about every index in your object model. It is true that indexes speed up value-based access to objects. But keeping the index up-to-date slows down write operations. You must find the optimal number of indexes for your application to balance these two conflicting aspects.

Often, it is difficult to anticipate the optimal index strategy for an application during the design phase. FastObjects provides a powerful schema versioning functionality that allows you to make changes to the schema (including indexes) even after product deployment. This gives you the opportunity to fine-tune your indexes using real-life data.

For more information on index definition, maintenance and optimization please refer to section 9 of the *FastObjects™ Administration Guide*.

3 Example Object Model

The proper design of the object model is at the heart of any well-running application. To assist the process of designing the object model, a graphical representation of the object model can be created using a CASE tool. Among the case tools that can automatically generate a FastObjects database schema are Rational Rose and OEW. The following is a sample object model that will be referenced throughout this paper.



The object model represents a School Enrollment System. The available courses have an id to uniquely identify them, a name and it is noted at which day in the week at which time they are taking place. They reference the instructor that will teach the course and the students that attend this course. They also reference the room in which they will take place. Each room has its name and the capacity noted. Every person involved in the school enrollment has a first name, a last name, its birthday and its gender as members. They also have a reference to an image and to an address object. This address object holds the street, the zip code, the city and the state where the person resides. A person can be a student or an instructor. An instructor can teach one or more courses. The courses an instructor actually teaches are referenced by each Instructor object. A student can be enrolled in one or more courses. The courses a student actually attends are referenced by each Student object.

In the following sections we will present and discuss various features that FastObjects provides. Based on the school enrollment example, above, we will show when using a specific feature is advisable and when it is not recommended.

4 Optimizing the Database Schema

FastObjects gives you complete freedom in designing your object model. Nevertheless, with regard to performance, you should consider some things when creating your object model. The proposals we present in the next sections are by no means to be interpreted as limitations to your

freedom to design your object model to fit your needs. But when performance is essential, you should be aware of the following in your object model designas a way to help FastObjects run at its optimum.

4.1 Avoiding Redundant Information in the Object Model

In general you should avoid having redundant information in your object model. On the one hand, this makes your object model more complex than it needs to be and on the other hand it results in more objects that must be written to or retrieved from the database.

There are, of course, special situations where redundant information could otherwise help avoiding frequent database accesses. If, for example, the sum of an invoice is stored in the database, it could prevent the necessity to iterate every invoice record each time its sum is needed.

4.2 Use a Minimum Number of Classes in Your Object Model

One thing you should consider during design phase is the granularity of your object model. For every object in the database FastObjects has to perform maintenance efforts. The object gets an own object identity, a special collection type to hold all objects of a specific class, called an extent, is maintained, relationships among objects must be maintained, etc.

You should carefully think about whether the objects you are going to design are really objects that need to have an own identity in your application or whether they are objects that have only a "value character". This means, these objects only make sense in a certain context and will be accessed solely through another specific object. In this case, the object has no stand-alone character, it depends on a specific other object. You should think about integrating those objects in the objects they depend on or to model them as embedded objects, so-called second-class-objects. You will find an in-depth discussion of second-class-objects later in this article.

4.3 Declare Which Object Data-members Really Need to Be Persistent

You should avoid making data-members persistent that are transient in nature. If you make these data-members persistent, your objects will be larger than necessary and writing and reading will take longer.

Imagine in our School Enrollment Example we have a data-member in each `Person` object to represent the age of that specific person. It does not make any sense to store this data-member because the age of a person depends on the current date. You should instead define the `age` data-member as `transient` and compute its effective value at the time the `Person` object is requested. For such purposes JDO provides the `InstanceCallback Interface`. You can use its `jdoPostLoad()` method, which will be executed just after loading of the objects data to calculate the age of the `Person` by subtracting birthdate from the current date.

4.4 Using Backward References With Indexes Instead of Large or Changing Collections

The standard Java collections are optimized for memory operations. These collections load all item references into memory at once and construct hollow objects for every referenced item. This causes performance problems when collections containing a large number of objects are used as class members.

If the application's object model contains a class which holds a large collection of references to other objects, and elements of the collection are frequently being added and removed, the continuous resizing of objects of this class can slow down your application. In such cases it is not advisable to directly express such a relationship in the application implementation by defining a collection of object references as a member of the respective class.

Instead, it is a good alternative to express this relationship more indirectly, keeping indexed backward references from the elements to their "owners" instead of keeping forward references from the collection to its elements. The ability for the object to recall all of the (former collection) elements can be implemented using FastObjects' extents or query capabilities finding the respective objects at the time when they are needed.

FastObjects is able to maintain indexes on object identity. This feature allows the placement of an index on an object or a collection of objects, and is recommended when requiring fast retrieval using a query on attributes of an aggregate relationship.

For instance, in our School Enrollment Example, consider the number of students participating in a particular course. Such a collection can become very large. Since we designed our model to keep a collection of all participating students in every `Course` object this can lead to poor performance when accessing `Course` objects.

```
public class Course {
        long id;
        String name;
        String dayTime;
        java.util.HashSet studentSet;
};
```

It may be a good solution to only hold a collection of references to the `Courses` in every `Student` object and define an index on the collection elements.

```
public class Course {
        long id;
        String name;
        String dayTime;
};
```

```
public class Student {
        java.util.HashSet courseSet;
};
```

Additionally an index on the object identities of the `courseSet` elements is defined in the `Student` JDO metadata file:

```
<class name="Student">
    <extension vendor-name="FastObjects" key="index"
                value="CourseIndex">
        <extension vendor-name="FastObjects" key="member"
                    value="courseSet"/>
    </extension>
</class>
```

If you need to look for all students participating in a particular course, you need only to set a filter on the `Student Extent` or to perform a query. The index allows filtering and accelerates a query for all `Student` objects that hold a particular `Course` object.

Using an indexed extent, for example, finding all students participating in a particular course `course` can now be done by defining an appropriate iterator:

```
// assuming pm represents the current PersistenceManager
Transaction txn = pm.currentTransaction();
txn.begin();
Extent students = pm.getExtent(Student.class,true);
// get an iterator of all students participating in a particular course
// using the CourseIndex index
// assuming course is the Course object we search all students for
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(students,"CourseIndex",course,true);
// do something useful with the iterator
. . .
txn.commit();
```

4.5 Using Backward References Instead of Queries

In our School Enrollment System, suppose the application requires that all the courses be listed that take place in a particular room named "A-1001". Since our object model only includes references from courses to rooms but not the other way around, we have to use a query to find all courses being held in a specific room. A query can be constructed through the `Course` extent.

The query in JDOQL would look like:

```
Extent extentCourse = pm.getExtent(Course.class,true);
Query aQuery = pm.newQuery();
aQuery.setFilter("room.name == \"A-1001\"");
aQuery.setCandidates(extentCourse);
aQuery.setClass(Course.class);
java.util.Collection result = (java.util.Collection) aQuery.execute();
```

The requirement to list all the courses in a particular room can also be achieved by programmatically maintaining a collection of backward references from the `Room` to the `Course` class. The drawback to maintaining a backward reference collection is that the data integrity must be ensured programmatically and two objects must be modified for each assignment. On the other hand, if the number of courses is great but only very few of them actually take place in a particular room, maintaining a collection of backward references may improve performance.

```
public class Course {
        . . .
        Room room;
};

public class Room {
        . . .
        java.util.HashSet courses; // Set of references
};
```

4.6 Using Embedded Objects

In our School Enrollment Example, the `Person` class has a reference to an instance of `Address`. In this model, multiple `Person` objects may share a single `Address` instance. However, if addresses are not shared between different persons, a performance gain can be had by *embedding* the `Address` instance. Embedded objects are not referenced but are a part of the containing object. In JDO, an embedded object is called a second-class-object or SCO. The containing object is a first-class-object or FCO. FCOs have object identity, SCOs do not.

```
class Address { // Address does not have to be persistence capable
        . . .
        String street;
        long zipCode;
        String city;
        String state;
}
```

If a `Person` instance never has more than one address, you could embed the `Address` object directly, as a single instance:

```
class Person {
        . . .
        Address address;
}
```

You declare objects as embedded by declaring the whole class as embedded.

Definition in the `Address` JDO metadata file:

```
<class name="Address">
<extension vendor-name="FastObjects" key="embedded" value="true">
    </extension>
</class>
```

The basic difference between embedding an object or an object collection and referencing an object or an object collection is, that embedded objects don't have any object identity. Even if the embedded object is an instance of a persistence capable class, the embedded object is stored without an object ID and will not be found in the class extent. If the container object (a `Person` object in the example) is deleted, any embedded objects are deleted as well because they are a part of the container object. An embedded object cannot be shared between different container objects, as it belongs to its container object.

Apart from a logical mapping, embedding objects into container objects can help to increase database performance:

- When a container object is loaded in memory from the database, the embedded objects are automatically retrieved as well. They are not handled as separate objects that are referenced outside the container object. Embedded objects are correctly reflected in the logical database storage as well—they are not stored in a different cluster but are stored together with the container objects.
- In object models where embedded objects are always required by the container object, retrieving embedded objects together with the container objects will help to improve performance because less IO communication is required. This becomes even more obvious in a network environment where every network call is expensive.
- Embedded objects do not have the "persistence overhead" of persistent objects on disk.
- No class extent is involved when storing or deleting the embedded objects.
- You do not have to assign embedded objects to the database.
- Locks cannot conflict across embedded objects.

4.7 Using Java Collections As Data-members

FastObjects JDO Binding supports the following Java collections and maps: `Collections`, `Vector`, `List`, `LinkedList`, `ArrayList`, `Set`, `HashSet`, `SortedSet`, `TreeSet`, `Map`, `HashMap`, `Hashtable`, `SortedMap` and `TreeMap`.

You can use any of these classes as members of the classes in your object model. In FastObjects, collection data-members will always be treated as second-class-objects. This means, the collection itself will be embedded in the parent object. The elements of the collection on the other hand will by default not be embedded. When you make an object with a collection data-member persistent, the objects that the collection holds will also be made persistent due to the persistence-by-reachability principle. By default, they become independent objects with their own object ID and their membership in the collection is stored as reference. This implies that after storing, these objects will be reachable as any other "normal" persistent objects, an extent is maintained for their class, etc.

When an object containing a collection data-member is retrieved from the database, the collection is automatically reconstructed as a whole with its elements containing references to the collection objects. These objects will not be automatically read from the database, but for each of these objects a hollow object will be created. The data of these objects will be retrieved from the database at the time when the object is visited. When iterating such a collection, one database call for every visited object will be performed.

One exception to this rule is when you have overwritten the `equals()` and `hashCode()` methods of the collection and use them when adding new elements to the collection. In this case all existing collection elements will be retrieved at once in order to be able to compare them.

4.7.1 Embedding the Collection Elements

If you do not need to access the collection objects other than through the collection, you should consider using collections with embedded elements. The objects contained in the collection will, in this case, not become independent objects but will be embedded in the collection. This implies that they will have no object ID, no extent will be maintained, etc. They will only be reachable through the respective collection. When an object containing a collection data-member with embedded elements is retrieved from the database this means, the collection with all its elements altogether with their data will be created at once.

This can lead to significant performance gains for both storing and reading objects with collection members. FastObjects need not maintain the object identities for all the collection objects and all objects will be read at once, which can be an advantage especially in client/server environments.

But, if the collection grows large, embedding its elements can lead to a performance loss, because the application must wait until all objects are read. This may slow down your application without having any advantage if you do not need all of the collection objects. And, also remember, embedded collection elements can not be referenced from anywhere

else, they only exist in the collection and are only accessible through the collection.

In our School Enrollment Example suggest a person could have more than one address. You can use a collection data-member to hold all addresses for every person. For example, a `HashSet` of `Address` instances.

```
class Person {
        . . .
        HashSet addressSet;
}
```

Collections are rarely, if ever, referenced by multiple objects. In all but exceptional cases, a collection object is the sole “property” of a single referencing object. Therefore, by default, a collection is a second-class-object in FastObjects for JDO. In the example above, a `HashSet` of `Address` objects are directly embedded in the container class `Person`. In this case, it is the `HashSet` object that is embedded in the `Person` object.

For embedded collections, the default behavior is that the collection elements (the `Address` objects in the example) are normal persistent objects in the database, referenced by the collection. However, you can also instruct FastObjects to embed the collection elements. This makes the entire collection, including the objects in the collection, an embedded part of the containing object. You declare elements in a collection as embedded the same way you declare objects as embedded by defining the whole class as embedded.

Definition in the `Address` JDO metadata file:

```
<class name="Address">
    <extension vendor-name="FastObjects" key="embedded" value="true">
    </extension>
</class>
```

4.7.2 Implementing Your Own Collection Class

You can prevent the collection from being stored as a second class object by writing your own collection class. This can be a simple wrapper class for one of the standard collection classes, for example.

```
class MyHashSet implements Set
{
    private HashSet hashSet = new HashSet();

// you must implement all Set methods: add, addAll, clear,
// contains, containsAll, equals, hashCode, isEmpty, iterator, remove
// removeAll, retainAll, size and toArray
// Example:
    public boolean add(Object o)
    {
        return hashSet.add(o);
    }
. . .
}
```

When you retrieve the parent object, only a hollow object for the collection is created and the time for reconstruction of the collection and creation of the hollow (collection element) objects is saved. This can be effective when the parent object contains other data, besides the collection, that might be needed more often then the collection itself. On the other hand, this approach means that the collection object will appear in the database as an independent object with its own object ID and will be accessible from anywhere, not only through its parent object.

4.7.3 Creating Your Own `java.util.Map` Implementation

When using large maps, you run into the same problems as with using other large collections. An alternative approach for maps is to create your own `java.util.Map` implementation and a helper class, that stores data triples: the owner object (this is the object the map belongs to), the key and the value of the map entry. Also, you have to define a compound index on the owner and the key. If you want to read one of the map entries, you can do a `findKey()` on the owner and the key and return the value. The drawback of this approach is that map entries will only appear in the map after commiting the transaction where they were created. They are not searchable immediately after creation but only when they have been written to the database.

Suppose in our School Enrollment Example we want to maintain a list of seminar papers for every student and want to be able to find them by keywords. We can implement this using a `Map`.

```
class Student
{

. . .
Map seminarPapers;

}
```

This map can grow large, if many keywords are applied to every seminar paper. An own Map implementation could look like this:

The helper class:

```
class MyMapImpl
{
    private java.lang.Object owner; // the Person object
                                    // the map belongs to
    private String key;   // we want to use keywords as keys,
                          // therefore we define the key as string
    private java.lang.Object value;

// you must implement the set and get methods for the private members
. . .
}
```

The compound index MyMapImplIndex on the MyMapImpl class, specified in the MyMapImpl JDO metadata file:

```
<class name="MyMapImpl">
    <extension vendor-name="FastObjects" key="index"
                value="MyMapImplIndex">
        <extension vendor-name="FastObjects" key="unique"
                    value="false"/>
        <extension vendor-name="FastObjects" key="member"
                    value="owner"/>
        <extension vendor-name="FastObjects" key="member"
                    value="key"/>
    </extension>
</class>
```

The own map class:

```
class MyMap implements java.util.Map
{
// constructors
// this constructor must be called in the constructor
// of the Student object
public MyMap() {}
// you must implement all Map methods: clear, containsKey,
// containsValue, entrySet, equals, get, hashCode, isEmpty, keySet,
// put, putAll, remove, size and values
```

```
// The get method
    public java.lang.Object get(String key)
    {
// get the current persistence manager
        javax.jdo.PersistenceManager pm =
              javax.jdo.JDOHelper.getPersistenceManager(this);
// get the MyMapImpl extent
        javax.jdo.Extent myMapImpls = pm.getExtent(MyMapImpl.class,true);
// get an MyMapImpls iterator using the MyMapImplIndex
        java.util.Iterator iter =
             com.poet.jdo.Extents.iterator(myMapImpls,"MyMapImplIndex");
// get the requested object by findKey()
// using the owner and the map key as keys
        boolean b = com.poet.jdo.Extents.findKey(iter,
             new java.lang.Object[]{this,key});
// return the map value if found
        if (b)
        {
            return ((MyMapImpl)
                     com.poet.jdo.Extents.current(iter)).getValue();
        }
        else
        {
            return null;
        }
    }

// The put method
    public java.lang.Object put(String key,java.lang.Object value)
    {
// get the old value object for return purposes
        java.lang.Object oldValue = this.get(key);
// delete the old map entry
        this.remove(key);
// set the new value object
        MyMapImpl myMapImpl = new MyMapImpl();
        myMapImpl.setOwner(this);
        myMapImpl.setKey(key);
        myMapImpl.setValue(value);
        javax.jdo.PersistenceManager pm =
           javax.jdo.JDOHelper.getPersistenceManager(this);
        pm.makePersistent(myMapImpl);
        return oldValue;
    }
}
```

A new map entry can now be set in the following manner:

```
// assuming student is the Student object where we want to put
// a seminar paper "paper" and "JDO" is the keyword
SeminarPaper oldpaper = (SeminarPaper)
                          student.seminarPapers.put("JDO",paper);
```

If you need to search for a seminar paper with a specific keyword, you can do this in the following manner:

```
// assuming student is the Student object where we search
// for a seminar paper and "JDO" is the keyword
SeminarPaper paper = (SeminarPaper)student.seminarPapers.get("JDO");
```

4.8 Not Using Non-persistent Capable Data-members

Non-persistence capable class types such as `GregorianCalendar` can be stored in a FastObjects database when they are serializable. This is sometimes necessary but you should think carefully about using this feature.

Serialized object (sub-)networks can only be written and read as a whole. This may cause heavy network transfer and time consumption during read and write operations.

Persistent object identity is preserved only for objects of persistence-capable classes. Objects of non-persistence-capable classes have no persistent object identity and are stored as separate instances in a serialized object network. Therefore when two or more persistent objects, referencing the same non-persistent object network are stored, the referenced non-persistent object network is serialized as two separate networks. When the persistent objects are read, memory contains two instances of the non-persistent network.

Members of serialized object (sub-)networks cannot be queried and cannot be indexed, thus filtering and searching methods are not available.

5 Using Transactions

Transactions are series of operations that succeed or fail as a unit. Transactions allow you to tell FastObjects to make changes to a database tentatively and to save or abandon any series of changes as a whole. In this sense transactions are responsible for data integrity. When using the FastObjects JDO Binding, you always need to use transactions for accessing objects from the database.

Any persistent object undergoes a certain lifecycle. At one point in time it is created, then it is stored, eventually is retrieved back, modified, written back to database one or more times, and finally is deleted. This lifecycle can be described in terms of the states of the object together with well defined state transitions. The object's state describes how the object behaves and the state transition describes from which state to which other state an object can change.

The state of an object depends on two aspects: 1) what was the object's state before state transition, and 2) what operation was done on that object. The second aspect contains all explicit operations on an object, like reading or modifying data-members and implicit operations on an object caused by transaction operations like `begin()` or `commit()`. The transaction performs implicit operations on the objects and changes the states of the objects to control that all objects in a transaction will be written to the database at once or not at all.

It is important to understand the object lifecycle and how it is influenced by transactions to understand how objects behave in your application.

5.1 Object Lifecycle

The JDO specification defines seven required states and the associated state transitions. At any point in time a persistent object has exactly one of the states described below.

5.1.1 Object Lifecycle States

The seven required states of an object are: transient, persistent-new, persistent-dirty, hollow, persistent-clean, persistent-deleted and persistent-new-deleted. These are described in the following sections.

5.1.1.1 Transient

When an object of a persistence capable class (i.e., objects of this class can be made persistent) is in transient state, it resides only in the memory of the Java Virtual Machine and behaves like any other object of any Java class, persistence capable or not.

5.1.1.2 Persistent-new

An newly constructed object of a persistence capable class is made persistent in the current transaction using `PersistenceManager.makePersistent()`. Such objects are in the state persistent-new. They reside only in the memory of the Java Virtual Machine. They have an object identity and, because of persistence-by-reachability, objects that are reachable from these objects are also given identity.

5.1.1.3 Persistent-dirty

Persistent-dirty objects are objects that represent persistent data that was changed in the current transaction.

5.1.1.4 Hollow

Objects that represent persistent data, but whose data-member values have not been read from the data store, are in a hollow state. Hollow objects have their object identity loaded, but not the values of their persistent data-members. The hollow state provides for the guarantee of uniqueness for persistent objects between transactions.

Objects committed in a previous transaction, returned by iterating an `Extent`, returned in the result of a query execution or navigating an object reference are also initially hollow.

5.1.1.5 Persistent-clean

Objects that represent persistent data and whose values have not been changed in the current transaction, are persistent-clean. This is the state of an object whose data-member values have been retrieved from the database have not been changed in the current transaction.

5.1.1.6 Persistent-deleted

Objects that represent persistent data and that have been deleted in the current transaction are persistent-deleted.

5.1.1.7 Persistent-new-deleted

Objects that represent objects that have newly made persistent and subsequently deleted in the same current transaction are persistent-new-deleted.

5.1.2 Object Lifecycle State Transitions

There are about fifty defined state transitions in the JDO specification. We will only treat a few important ones that occur during typical operations on objects. Please refer to the JDO specification to learn about the other state transitions.

5.1.2.1 Creation of Objects

When you first create an object of a persistence capable class, this object is not persistent but rather in the transient state and behaves like any other Java object.

5.1.2.2 Storage of Objects

When you make an object an object persistent using the `makePersistent()` method of the `PersistenceManager`, the object state changes from transient to persistent-new.

This does not mean that the object is written to the database. It is only marked that it shall be written to the database. The actual writing occurs upon successful transaction commit and the object's state is changed to hollow.

5.1.2.3 Retrieval of Objects

By default, objects will always be retrieved from the database in the hollow state. This means that an object will be constructed in memory that represents the object stored in the database but is not filled with its data. When the first of the object data-members is accessed, the data will be retrieved from the database and the object's state is changed to persistent-clean.

5.1.2.4 Modification of Objects

If one or more of the object data-members is changed, the object's state is changed to persistent-dirty. This marks the object as modified which indicates to the transaction that the object must be written back to the database at transaction commit.

5.1.2.5 Deletion of Objects

A persistent object can only be deleted using the `deletePersistent()` method of the `PersistenceManager`. At the time the method is called, the object's state is changed to persistent-deleted. The actual deletion takes place only on transaction commit.

5.2 The Transaction Cache

FastObjects has an automatic transaction cache that minimizes read and write operations, as opposed to JDBC™ and relational databases, where the application developer is responsible for the read/write operations, and Java Serialization, where object networks are always read and written completely.

Each persistent object has an object ID. This ID is used by the database to uniquely identify the object beyond its lifetime in transient memory.

The application developer usually does not need to worry about object IDs. Although IDs may be used to retrieve objects from the database, the preferred approach is to retrieve objects by walking through extents, by executing queries, or by direct navigation (traversing references). The transaction cache maintains a table for mapping object IDs to objects. This table is used to reconstruct references when objects are loaded from the database, and to make sure that every object is loaded only once. Consider the case where two parent objects point to the same child object, and the two parent objects are read independently. When reading the second parent object, the transaction cache figures out that the child object has already been constructed and the parent-to-child reference is set accordingly.

The database holds references only to application objects within its transaction cache. The database never references an application object outside a transaction.

An object may enter the transaction cache in the following ways:

- A `makePersistent()` operation is executed with the object as an argument.
- A reference to this object is read from the database.
- The object becomes persistent due to persistence by reachability.
- A reference to an object is reused in a new transaction when both the old and new transactions were created as part of the same persistence manager.

Thus, transient objects that have just been allocated by a new operation are not considered part of the transaction cache.

An object leaves the transaction cache when:

- The transaction is terminated (by committing or aborting the outmost level).
- An object residing in the transaction cache is eligible for garbage collection if no application helds any strong reference to it. If such an object is garbage collected it leaves the transaction cache.

5.3 Reading and Writing Objects in Transactions

When you read an object from the database, the transaction obtains a read lock for the object. This tells FastObjects that the object is in memory and that it is being viewed by this transaction. Attempts by other transactions to make changes to this object need to wait until the object is no longer being viewed.

When you modify the object, the transaction attempts to obtain a write lock. The write lock informs the FastObjects database engine that this object is going to be modified, and that no other transaction should see it until the changes are written back to the database.

It is important to keep this automatic locking in mind when working with objects in transactions. Each time you read an object, the transaction attempts to lock it. When you modify an object the transaction first attempts to obtain a write lock on the object. This can fail if the object is being viewed (a read lock exists) in one or more other transactions. This then results in an exception.

The locks are obtained at the moment the object is read or is written but the read locks are not released until transaction commit (or rollback).

5.4 Transaction Duration

5.4.1 Long Lasting Transactions

Long lasting transactions can become a performance problem when they are holding locks to objects that other transactions try to write. As long as one or more transactions hold read locks to an object, no other transaction is allowed to modify this object. And, even worse, when you modify an object, the transaction obtains a write lock, so that no other transaction can access that object for either reading or writing.

You should try to narrow the duration of your transactions. This is especially true for transactions where objects are written. You should check if it is possible to split transactions. Use longer lasting transactions only for reading objects and shorter transactions that write objects.

The `PersistenceManager` keeps references to objects between two transactions. Therefore if you use the same `PersistenceManager` you can use the objects you load in one transaction in a new transaction. Here is an example:

```
// assuming pm the current PersistenceManager
Extent nodes = pm.getExtent(Node.class,true);

Transaction txn1 = pm.currentTransaction();
txn1.begin();
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(nodes,“NameIndex”);
boolean found = com.poet.jdo.Extents.findKey(iter,“root”);
```

```
Node root = (Node) com.poet.jdo.Extents.current(iter);
Node subTree = root.getChild(1).getChild(2).getChild(3);
txn1.commit();

Transaction txn2 = pm.currentTransaction();
txn2.begin();
subTree.name = "0.1.2.3";
. . .
txn2.commit();
pm.close();
```

Note that in the intermediate step, after committing one transaction and before beginning the new one, FastObjects is quiet. It does not keep strong references or locks in memory. This means that another process or client or thread may have altered the objects. For this reason, the object's data is refreshed when it is accessed in the following transaction.

5.4.2 Short Transactions

When your applications consists of numerous fine-grained transactions, it may be difficult or require too much memory to navigate to a particular object.

If you use the same `PersistenceManager`, the FastObjects JDO Binding automatically checks when the object of a former transaction is accessed for the first time in a new transaction. If it can be reused, FastObjects refills the object with valid data from the database. Checking is done because the object might have been changed after the first commit. For example, it checks whether the transaction to which the passed object was assigned has been deleted and whether it belonged to the same database. If everything is correct, the object enters the transaction cache of the new transaction.

5.5 Transaction Size

5.5.1 Large Transactions

Large transactions are transactions that contain many objects. Such transactions can require large amounts of memory. The memory consumption during the lifetime of the transaction, and especially during commit, increases with the number of modified objects. If you are using very large transactions, you may want to see if it is possible to split them. Alternatively you may use the solutions described in *Using Checkpoints* and *Using Autoflush during Commit.*

Transactions cache all objects that were modified until the transaction is committed or rollbacked to ensure data integrity. The FastObjects transactions are designed to be scalable with respect to the number of objects involved, so it is not a problem to have thousands of objects involved in a single transaction. The drawback is memory consumption.

For this reason, the FastObjects JDO Binding provides operations to intentionally remove objects from the transaction cache: `PersistenceManager.evict()` to remove a single object from the cache, and `PersistenceManager.evictAll()` to remove several objects at once.

Objects can be evicted only if they have not been modified (i.e., its state is not Dirty). Therefore, objects are typically evicted after a checkpoint operation, which flushes their content, clears the dirty flag and sets the object's state back to Hollow.

The strategy of using large transactions, using checkpoints to flush data while maintaining the transaction cache (see section Using Checkpoints in Transactions), and occasionally evicting objects is often applied when browsing data. Selecting the candidates that should be evicted is not always easy, especially if complex object network structures are involved. But in cases such as walking linearly through an extent, the strategy is optimal:

```
// assuming pm the current PersistenceManager
Extent students = pm.getExtent(Student.class,true);
Transaction t = pm.currentTransaction();
t.begin();
java.util.Iterator iter = students.iterator();
while(iter.hasNext())
{ // assuming you have defined a toString() method for Student
    Student student;
    student = (Student) iter.next();
    System.out.println(student);
    pm.evict(student);
}
t.commit();
pm.close();
```

5.5.2 Small Transactions

However, having many very small transactions may result in poor performance. Starting a transaction implies some administration overhead in FastObjects and is always accompanied by calls to the database. These are time consuming operations—especially in client/server environments. The same is true for commit or rollback of a transaction, even if no objects have been modified. If you are running very small transactions with very few objects, these administration operations may affect the overall performance.

5.6 Commit Time of Transactions

The time required for committing a transaction increases with the number of objects contained in the transaction. This is especially true for client/server environments. Also, the need to maintain many indexes can increase the commit time.

5.7 Using Checkpoints in Transactions

FastObjects supports transaction checkpoints to reduce memory consumption during commit. Checkpoints mirror the "File Save" menu item of typical applications. They flush all changes to the database but keep everything open. In other words, they maintain the transaction cache and all locks.
Checkpoints specify that an intermediate commit should be performed for the objects being stored within a transaction. This means that the transaction is no longer atomic, i.e. the transaction no longer acts like a single all-or-nothing transaction, but is subdivided into smaller transactions bounded by the checkpoints. The intermediate checkpoint calls perform real commits and cannot be rolled back. However, for long transactions that involve many objects, checkpoints can help to avoid running out of memory. A checkpoint can be performed by calling the `com.poet.jdo.Transactions.checkpoint()` function. When you call the `checkpoint()` method, the modified objects associated with the transaction are written to the database, but they maintain their locks. They remain accessible after the checkpoint and will not be reloaded when their fields are accessed after the checkpoint. In all other aspects, calling the `checkpoint()` method is equivalent to calling the `commit()` method of a transaction followed immediately by the `begin()` method of a new one.

5.8 Using Autoflush During Transaction Commit

There is another possibility for decreasing memory consumption during commit of large transactions. Using the autoflush transaction property, the commit operation is internally done within a sequence of nested transactions. This decreases the memory consumption but also increases the execution time of the commit operation.

The following example shows how to enable autoflushing during commit:

```
Transaction t = pm.currentTransaction();
java.util.Properties p = com.poet.jdo.Transactions.getProperties(t);
p.put("shadowTransaction","true");
p.put("autoFlush","10000");
com.poet.jdo.Transactions.setProperties(t,p);
t.begin();
// Do something useful
. . .
t.commit();
```

In this example, the commit internally uses nested transactions with 10000 objects each. Try out different values of the autoflush property, to find the optimum ratio between memory consumption and runtime in your application.

5.9 Using Objects Outside of Transactions

Objects that participate in a transaction are, by default, inaccessible after the transaction terminates. When a data-member of such an object is accessed, an exception is thrown. This can be considered a harsh

restriction. It can be weakened by setting the transaction property `retainValues` to `true`. The setting of this property indicates that eviction of the persistent objects does not take place after transaction commit. The state of the objects is changed to Persistent-Nontransactional. You can read and modify data-members of those objects outside a transaction as long as they are not accessed in a new transaction. Of course, outside of a transaction the database connection is lost and any changes to data-members will not be reflected in the database.

You can only access objects outside a transaction, that were retrieved in the transaction. Objects in the Hollow state can neither be read nor written.

Nevertheless, it is sometimes useful to retain objects and have them accessible after the transaction has ended. In this case, you must make sure that the objects have been properly retrieved.

```
// assuming pm the current PersistenceManager
Extent students = pm.getExtent(Student.class,true);

Transaction t = pm.currentTransaction();
t.setRetainValues(true);
t.begin();
java.util.Iterator iter = students.iterator();
Student student = (Student) iter.next();
pm.retrieve(student);
t.commit();
. . .
// assuming you have defined a toString() method for Student
System.out.println(student);
pm.close();
txn.begin();
```

Note: The above sample code uses `retrieve()` to explicitly ensure the object is retrieved from the database. This is necessary because the attempt to access an object outside a transaction that was in the hollow state at transaction commit will throw an exception. But even `retrieve()` may not retrieve the object in all circumstances. If the object is locked by another transaction `retrieve()` will throw an exception and leave the object in hollow state.

5.10 Reusing `PersistenceManager` and `PersistenceManagerFactory` Objects

In FastObjects, each `PersistenceManagerFactory` maintains a database connection to the underlying data store. The database connection is established when the first `PersistenceManager` is returned from the `PersistenceManagerFactory`. The connection is closed when the last `PersistenceManager` of a

`PersistenceManagerFactory` is closed. Since opening and closing database connections is time consuming, especially in client/server environments, the number of these operations should be optimized.

If possible, each client should have only one `PersistenceManagerFactory` for each database. `PersistenceManager` instances should be obtained from this specific `PersistenceManagerFactory`.

To avoid the database connection being closed and opened again and again, the application should ensure that at least one `PersistenceManager` remains active for as long as operations are expected on the database. In the following example, the database connection is closed within each `PersistenceManager.close()` call and reopened during `PersistenceManagerFactory.getPersistenceManager()`.

```
PersistenceManagerFactory pmf =
    com.poet.jdo.PersistenceManagerFactories.getFactory();
pmf.setConnectionURL("FastObjects://LOCAL/base");

// assuming finished() represents a condition to end the loop
while (!finished())
{
    PersistenceManager pm = pmf.getPersistenceManager();
// Do something useful
    . . .
    pm.close();
}
```

Reusing the same `PersistenceManager` avoids repeated close/open operations:

```
PersistenceManagerFactory pmf =
    com.poet.jdo.PersistenceManagerFactories.getFactory();
pmf.setConnectionURL("FastObjects://LOCAL/base");
PersistenceManager pm = pmf.getPersistenceManager();

// assuming finished() represents a condition to end the loop
while (!finished())
{
// Do something useful
    . . .
}
pm.close();
```

Alternatively, you may specify, that the database connection is not closed implicitly, when the last `PersistenceManager` is closed.

This can be done with the `PersistenceManagerFactories.setExplicitClose()` method. In this case, you have to call the

`PersistenceManagerFactory.close()` method to explicitly close the database connection.

```
PersistenceManagerFactory pmf =
    com.poet.jdo.PersistenceManagerFactories.getFactory();
pmf.setConnectionURL("FastObjects://LOCAL/base");
com.poet.jdo.PersistenceManagerFactories.setExplicitClose(true);

// assuming finished() represents a condition to end the loop
while (!finished())
{
    PersistenceManager pm = pmf.getPersistenceManager();
// Do something useful
    . . .
    pm.close();
}
pmf.close();
```

When you are using the same objects in a sequence of subsequent transactions, it is also a good idea to reuse the `PersistenceManager`. In this way, the object instance of the previous transaction can be reused, and no new instance has to be created.

6 Avoiding Unnecessary Read and Write Operations

6.1 Automated Management of Reading and Writing Objects

In the FastObjects JDO Binding, objects are read from the database only when at least one of their data-members is accessed, and they are written back to the database only when they have actually been modified. If an object reference is passed to the application, the reference points to an object in the Hollow state, i.e., an object of the proper class whose memory has been allocated but not yet filled with data from the database. When the application then accesses a data-member, a shared lock is acquired (if concurrency control is enabled) and the actual state is changed to Persistent-Clean (the object is retrieved).

```
PersistenceManagerFactory pmf =
    com.poet.jdo.PersistenceManagerFactories.getFactory();
pmf.setConnectionURL("FastObjects://LOCAL/base");
PersistenceManager pm = pmf.getPersistenceManager();
Extent students = pm.getExtent(Student.class,true);
Transaction t = pm.currentTransaction();
t.begin();
java.util.Iterator iter = students.iterator();
```

```
// returns hollow object
Student student = (Student) iter.next();
students.close(iter);
// student will be read/locked
System.out.println(student.name);
// returns hollow object
Address address = student.address
// address will be read/locked
System.out.println(address.city);
// write-lock on address
address.zipCode = 12345;
// address will be written
t.commit();
pm.close();
```

In nearly all cases, this transparent behavior is what the application requires, and it is more efficient than any handwritten code.

Nevertheless, you can test the object's state with the methods defined in the `JDOHelper` class. You can determine whether an object is a new object (`isNew()`), whether it represents a persistent instance (`isPersistent()`), whether it is associated with the current transaction (`isTransactional()`) or whether it is dirty (`isDirty()`). It is also possible to explicitly retrieve an object or make it dirty. This is seldom needed except in the following cases:

- When an object is accessed or modified using the Java Reflection API, or before serializing it
- When using the `retrieveAll()` method may speed up reading in client/server scenarios
- When objects are supposed to be accessed outside the transaction (see below)
- When you access arrays such as `Object[]`, `String[]` and so on.

The object model granularity and the modeling of redundant information discussed in the sections Avoiding Redundant Information in the Object Model and Use a Minimum Number of Classes in Your Object Model are the only parameters you as the application designer must be aware of. These parameters naturally influence the number of objects that must be written to or retrieved from the database.

6.2 Deleting Objects

In FastObjects there are, in principal, two ways to delete objects from a database, a direct way and an indirect way.

6.2.1 Directly Deleting Objects

It is best to delete objects directly as soon as they are not needed any longer. You should take care to also delete the referenced objects. You can use the `InstanceCallback` interface method `jdoPreDelete()`. Instance callbacks provide a mechanism for performing user-specified operations for specific JDO instance life cycle events. You implement the `InstanceCallbacks` interface in the same manner that you implement any interface in Java.

```
class Person implements javax.jdo.InstanceCallbacks
{
//...
String firstName;
String lastName;
//...
// the InstanceCallbacks methods
    public void jdoPostLoad()
    {
//...
    )
    public void jdoPreStore()
    {
//...
    }
    public void jdoPreClear()
    {
//...
    }
    public void jdoPreDelete()
    {
// get the current persistence manager
        PersistenceManager pm = JDOHelper.getPersistenceManager(this);
// delete the referenced address object
        pm.deletePersistent(this.address);

// delete the referenced image object
        pm.deletePersistent(this.picture);
    }
}
```

If your FastObjects JDO class implements the `InstanceCallbacks` interface, the FastObjects database engine automatically calls the methods defined by the interface. The `jdoPreDelete()` method is called when the application calls `pm.deletePersistent(obj)` and prior to the object actually being removed from the database.

```
// assuming pm the current persistence manager
// and person the Person object to be deleted
pm.deletePersistent(person);
```

Alternatively FastObjects provides a mechanism for directly traversing, or "walking", an object network starting from a given root object or collection of root objects and collecting the objects of the network. You can use the resulting collection to perform whatever operations you desire, i.e. collect and delete all dependent objects of a given object. The following example shows how this is accomplished:

```
// assuming pm the current persistence manager
Transaction txn = pm.currentTransaction();
txn.begin();
NetWalker walker = new NetWalker();
// tell the walker to collect all sub-objects
walker.setDeep(true);
// assuming person the Person object to be deleted
Collection dependentObjects =
    walker.compute(txn,person);
pm.deletePersistentAll(dependentObjects);
myTxn.commit();
```

6.2.2 Indirectly Deleting Objects

The second principal way to delete objects from a FastObjects database is to use the FastObjects Java garbage collector, PtGC. Garbage collection is a Java platform feature that removes objects from a database that can no longer be referenced by the application. You can run it from the command line or you can also call the `garbageCollection()` method of the class `com.poet.jdo.admin.DatabaseAdministration`.

There are a number of limitations that you should be aware of that can restrict the abilities of PtGC:

- Because PtGC potentially deletes objects from the database, it requires exclusive access. If any other clients are using the database, garbage collection is not possible.
- To identify the deletable objects, the garbage collector checks whether each object is reachable from any object bound to the database as a *named object* or you must explicitly mark classes as having non-weak instances by adding a weak entry under the appropriate classes section in the Java options files:

```
[classes\MyPackage.MyClass]
persistent = true
weak = false
```

This prevents instances of classes from being deleted by the garbage collector.

- Deleting objects using PtGC is slower than deleting objects from within an application because PtGC has to verify for every object that it can be deleted by following every reference to this object back to the root to find out whether the root object is a named object or not.

6.3 Retrieving Objects

As pointed out before, in general with FastObjects, objects are read from the database only when at least one of their data-members is accessed. You can, however, influence the point in time when objects are read from the database by using the techniques presented in the next sections.

6.3.1 Using FastObjects Active Java Cache

In some multi-user applications, e.g., Internet data-delivery applications, a large number of clients are busy accessing a single database. The clients in these applications are, for the most part, only reading the data from the database and presenting the data in some form to the application user. Multi-tier environments have been developed for distributing the work involved in delivering such large volumes of data to so many users. Typically, the database server handles data requests from a manageable number of "application servers". Each application server, in turn, handles the requests of many clients. From the point of view of the database server, each application server is a request funnel and data fan-out device. This hierarchical approach to the distribution of the data reduces the number of client connections to the FastObjects database server but not necessarily the number of requests to the FastObjects database server.

To reduce the number of requests arriving at the database server, the application server can be written to optimize its communication to the database server. One strategy is to provide data caching at the application server rather than at the database client side (which is standard in a straightforward client-server scenario). This is particularly effective if the numerous end clients that are communicating with the application servers are often requesting the same data. When you use FastObjects, caching at the application server is accomplished with the Active Java Cache.

The FastObjects Active Java Cache provides a mechanism for minimizing FastObjects database server requests in application scenarios where the bulk of the database server traffic is read-only. The cache keeps recently read objects locally, at the application server, so that the application server can deliver the object without re-reading the object from the database server. This can significantly reduce the database server traffic for frequently read objects.

The cache is most efficient for the high-volume reading of objects. However, the occasional need to update objects that may reside in one or more caches must also be met. An important component of the FastObjects Active Java Cache is a lock service that also caches the read-locks held by the end clients. This further reduces the database server traffic because the application server cache can grant the read lock to additional end clients that request the object without communicating with the database server. The lock service is notified when a write-lock request is issued to the database server and can be configured to revoke any read-locks that are still held by the end clients. Cache consistency is not a concern.

The FastObjects Active Java Cache can be used transparently by the application. There is no need to change application logic to use the application server cache.

Note that the configuration of isolation levels and lock mode mapping is not supported together with the Active Java Cache and lock server usage.

The application cache is activated and configured directly in your application code, before you open the database, by specifying the desired cache size.

The parameter is passed by a property object as follows:

```
Properties cacheprops = new Properties();
cacheprops.setProperty("databaseCacheSize","5000");
// assuming pmf is the PersistenceManagerFactory
PersistenceManagerFactories.setBackendProperties(
                  pmf,"fastobjects",cacheprops);
```

The database cache size specifies the number of cached objects and depends heavily on the configuration and profile of your machine and application. 5000 objects is just an example value. The cache can be switched off by specifying a cache size of 0.

For more information on the Active Java Cache and how to use it, please refer to the *FastObjects Programmer's Guide - Java™ Platform (JDO)*.

6.3.2 Using Access Patterns

One way to control when objects will be loaded into memory is to use *access patterns*. Many applications retrieve sub-objects following clearly defined patterns. The rules may even be different for certain modules of the application. Access patterns allow you to tell FastObjects—before requesting the root object—which associated referenced objects may also be needed. Access patterns only work in client/server architectures, telling the server which objects are transferred together with the root object. The referenced objects will not constructed as hollow objects but will be filled with data immediately. The network transfer for the desired parts of an object network is bundled and this can significantly reduce the time necessary to access each associated object as it is used in the application.

Suppose, in our School Enrollment example a browser screen may want to display persons together with their address. To print the data, the browser will always request the address after requesting a person.

Another module of the browser might display persons along with their picture, always requesting the image. Here the definition of both the `Image` and the `Address` object as references together with defining two different access patterns can be useful, defining exactly which sub-objects shall be pre-loaded for each task.

Access patterns are defined in the `[schemata]` section of the *server* configuration file (`ptserver.cfg`). The access patterns for loading `Address` or `Image` objects together with a `Person` object would look like this:

```
[schemata\PersonDict\accessPatterns]
usedPatterns = personAddress,personPicture
defaultPreloadDepth = 2
maxPreloadObjects = 7

[schemata\PersonDict\accessPatterns\personAddress]
pattern = *.Person.address

[schemata\PersonDict\accessPatterns\personPicture]
pattern = *.Person.picture
```

The access patterns that you specify in the configuration file are parsed when the FastObjects server physically opens a database. This is done when the first `PersistenceManager` is obtained from a `PersistenceManagerFactory`. To see how the server processes access patterns, this configuration file entry can be used:

```
[debug]
accessPatterns = true
```

To use access patterns you must activate them. Activating access patterns can be achieved in two different ways: automatically or from the application. If the access pattern definition in the `ptserver.cfg` configuration file defines a default access pattern, this access pattern is activated automatically. To activate an access pattern from the application, you have to set transaction properties:

```
Transaction txn = pm.currentTransaction();
java.util.Properties props = com.poet.jdo.Transactions.getProperties(txn);
props.put("accessPatternName","personAddress");
com.poet.jdo.Transactions.setProperties(txn,props);
txn.begin();
```

Using access patterns only makes sense in client/server architectures. If you try to use access patterns in `LOCAL` mode, they will have no effect. For more details about access patterns, refer to *FastObjects™ Programmer's Guide - Java Platform (JDO).*

6.3.3 Enable `immediateRetrieve` When Reading Objects From an Extent

In FastObjects 9.0 objects are retrieved from the database in hollow state by default. This means when accessing an object through the `Extent` iterator only an "envelope" object without the data is constructed in memory. The data is reloaded, when the first data member is requested. In client/server environments this results in two server calls for every accessed object, one to construct the hollow object and one to retrieve the data.

Setting the `immediateRetrieve` property of the `Extents` utility class to true causes all objects that are accessed through the `Extent` iterator to be constructed and filled with data at once. In client/server environments this means that only one server call is performed for every accessed object.

```
// assuming students is the extent of the Student class
java.util.Iterator iter = students.iterator();
Extents.setImmediateRetrieve(iter,true);
```

Note:

Immediately retrieving objects implies a change in the locking behavior. Objects retrieved in hollow state will not be locked until their first data memeber is requested. Objects retrieved in retrieved stete will be locked immediately at construction time.

The `immediateRetrieve` property can alternatively be enabled globally by using the switch

`-Dpoet.extents.immediateResolve=true`

when starting the Java Virtual Machine.

In FastObjects 9.5 the `immediateRetrieve` property is enabled by default.

In some cases it might be advisable to disable `immediateRetrieve`, either to realize a delayed locking or to prevent large amonuts of data to be loaded when it is not needed.

6.3.4 Using `setPreFetch()` When Reading Elements From an Extent

`setPreFetch()` is a function of the `com.poet.jdo.Extents` utility class which improves the speed of retrieving objects from the database.

```
public static void
setPreFetch(java.util.Iterator iter,
            int numberOfObjects)
```

Following is a typical situation for using this function: Assume that you want to display the first ten elements of an `Extent` in a list box. This means that the first ten elements of the `Extent` must be retrieved from the database. By calling the `setPreFetch()` method with an argument of 10 (`setPreFetch(iter,10)`) the ten objects will be retrieved from the database at once.

In a client/server environment, this means all objects requested with `setPreFetch()`will be transferred from the server into the client's memory in just one server call. This behavior will have much better performance than requesting each of the ten objects sequentially from the server, as it is the default behavior for retrieving objects from an `Extent`. `setPreFetch()` only makes sense in a client/server environment. It is ignored when operating in `LOCAL` mode.

To enable `preFetch` globally – e.g. for testing purposes – you can also use the switch `-Dpoet.extents.preFetch=10` when starting the Java virtual machine.

Note that the `immediateRetrieve` property of the `Extents` utility class is automatically set to “true” when using `preFetch` with a value greater zero.

For more information on `immediateRetrieve` refer to section Enable `immediateRetrieve` When Reading Objects From an Extent.

6.3.5 Using `retrieveAll()` to Load All Objects of a Collection

By default, FastObjects retrieves objects when the first of their data-members is accessed. In client/server environments this means a server call for every object. You can tell FastObjects to retrieve objects contained in a collection all at once using the `PersistenceManager.retrieveAll(java.util.Collection pcs)` method thus making only one server call for the collection contents.

You can also use the `PersistenceManager.retrieveAll(java.lang.Object[] pcs)` method to tell FastObjects to retrieve all objects contained in the objects array.

6.3.6 Using the `requires-extent` Keyword

In JDO, all persistence capable classes normally have a class extent. It is possible to turn off the extent maintenance for a class in the XML metadata using the class attribute `requires-extent`.

For each class extent, an index tree is maintained that is updated each time an object of that class is added, modified, or deleted. The `Extent` is necessary for the following two situations: 1) for iterating over all objects of a class and 2) for performing queries on all objects of that class. If an object will not be accessed through either of these mechanisms, then you should indicate that the class does not need to maintain an `Extent`.

For instance, in our School Enrollment System example, the `Person` class has a reference to an `Address` class. You can easily imagine that in your application you will probably do things like listing all the persons in the database. For such purposes the `Person` class needs an `Extent`. However, you will probably never perform tasks like listing all addresses without the persons they belong to, this simply makes no sense. The `Address` objects will always be accessed via an `Person` object, therefore the `Address` class does not need to maintain an `Extent` since all requests on the address will be through the person.

The following is the entry in the `Address` JDO metadata file, defining that the class should have no `Extent`:

```
<class name="Address" requires-extent="false" />
```

6.3.7 Avoiding Unnecessary Objects in Memory

When FastObjects reads an object from the database, it never creates a second copy of the object in memory for the same persistence manager. Instead a reference to this object is created. The Java garbage collector keeps track of the number of references made to each object. If there are no more active references, the object can be garbage collected.

Upon transaction commit, FastObjects, by default, automatically sets the object's state to hollow. But this does not mean the Java garbage collector can clear the memory. The reference is still held until either the lifetime of the object variable is completed or you explicitly set the object variable to another object or to `null`. Therefore you should take care to not expand the object's lifetime longer as it is necessary.

With `PersistenceManager`'s `evict()` and `evictAll()` methods you can give a hint to the persistence manager that you do not need a specific object in memory any longer. These methods immediately clear all references held by the object with the result that after completion of the transaction the object will be garbage collected. Regardless of whether the `retainValue` flag is set to `true` or not.

7 Choosing the Appropriate Access Method, Object Retrieval

7.1 Direct Navigation

7.1.1 Using Direct Navigation

An Object Database Management System maintains the semantics of the data model. This allows objects and their relationships to be retrieved through navigation. By contrast, a relational database system must recreate all relationships using joins. Developers more experienced with relational databases are inclined to initiate queries to retrieve objects, since they are unfamiliar with traversing relationships. Given a one-to-many relationship between two classes, an object database system allows the objects in the collection to be retrieved directly without the necessity of performing a query. The ability to navigate relationships using an object database system reduces the necessity for queries.

7.1.2 Using Extents for Fast Access

FastObjects' JDO Binding provides a special type of collection, called an extent, to retrieve objects. Class extents are special collections that are maintained by the database. These extent collections include all objects of a specified class type.

A class extent is a “virtual” collection of all or some instances of a so-called “candidate” class and its subclasses. It is a virtual collection

because it does not exist in memory, as is the case for classes such as `java.util.HashSet` or `java.util.Vector`. Therefore working with extents rather than working with standard Java collections can result in significant performance improvements. The objects, that are hold in the extent will not be loaded into memory unless they are explicitly requested. This means less memory consumption and in client/server scenarios less client/server calls. You can use the extent of a particular class together with searches, filters or queries to find and load only the objects you need.

FastObjects provides various search operations on extents which are often much faster than using queries. These operations can be divided into two groups:

- Finding objects by index keys. These functions can be used for search operations based on a specified index.
- Finding objects by setting extent filters. These functions are very similar to queries, but the result collection is computed while iterating the `Extent`. The query result collection will always be created at once, including the creation of hollow objects for every collection item. Filtering an `Extent` does not load any objects or create hollow objects. The objects only loaded when they are actually accessed.

7.1.3 Using Iterators on Extents: `advance`, `current`, `previous`, `reset`

In the JDO standard API, the support for iteration through an extent is restricted to the facilities provided by the interface `Iterator` (excluding the optional method `remove()`.)The order in which the elements are accessed is undefined (and differs for different database engines). The method `Extent.close()` allows you to free the resources of the iterator. In addition, FastObjects extends this API to provide additional functionality such as iteration in the order specified by an index, iteration forwards and backwards and skipping a certain number of elements.

When using iterators on extents, you should be aware that the largest administration effort for FastObjects is creating the iterator.

Creating an `Extent` object does not cause a database call because the `Extent` object will not be filled with any objects during creation. The objects will be retrieved when an iterator on the `Extent` is created and the objects are accessed via this iterator.

An iterator consumes some resources. For this reason, reusing iterators is a good idea. FastObjects provides some methods which allow iterators to be reused. This is an enhancement to the standard specification of iterators.

You can use the `previous()` method to navigate backwards through the iterator. This is also possible using an offset. And the `reset()` method repositions the iterator to the first object in the `Extent`.

In addition, two other methods are available for positioning the iterator. `advance()` positions the iterator forwards with a specified offset and `current()` returns the current object without advancing the iterator.

These methods add considerable flexibility and allow reusing iterators. This can speed up the performance of your application.

7.1.4 Reusing `Extent Iterators`

When using `Extent Iterators`, the following should be kept in mind:

- In client/server scenarios, for every `Extent.iterator()`call a server call is executed.
- Resources for the `Iterator` are held until the current transaction is terminated.

Therefore, in many cases, it is a better choice to reuse an existing `Iterator`. The last of the following three options is the most performant, especially if the `Extent` has only a few elements or only a small part of the `Extent` will be actually visited.

1. Option:

```
// assuming finished() represents a condition to end the loop
while (!finished())
{
    Extent ext = pm.getExtent(Person.class,true);
    java.util.Iterator iter = ext.iterator();
// do something useful with the Extent Iterator
    . . .
    ext.close(iter);
}
```

2. Option:

```
Extent ext = pm.getExtent(Person.class,true);
// assuming finished() represents a condition to end the loop
while (!finished())
{
    java.util.Iterator iter = ext.iterator();
// do something useful with the Extent Iterator
    . . .
ext.close(iter);
}
```

3. Option:

```
Extent ext = pm.getExtent(Person.class,true);
java.util.Iterator iter = ext.iterator();
// assuming finished() represents a condition to end the loop
while (!finished())
{
    com.poet.jdo.Extents.reset(iter);
// do something useful with the Extent Iterator
    . . .
}
ext.close(iter);
```

7.1.5 Using Indexes for Sorted Access

Programs often need to retrieve objects in a specific sorted order.. For example, in the School Enrollment System, the application may require that the students be displayed in alphabetic order. An index built on `Person.lastName` for the `Student Extent` will allow all students to be retrieved by this index as well as queries to be performed on the `Student Extent` with the result collection being returned in alphabetic order.

By default, `Extents` use the surrogate (OID) as the index sort order. This is equivalent to sorting the objects by the order in which they were first assigned to the database. The `com.poet.jdo.Extents.iterator()` methods can be used to obtain an `Extent Iterator` that uses a specified index.

An iterator for all instances of a class—in a particular—order can be obtained only if the database is configured to maintain an appropriate index.

To select an alternative index order for the `Extent`

1. Define the index in the database schema:

Index definition in the `Student` JDO metadata file:

```
<class name="Student">
    <extension vendor-name="FastObjects" key="index"
                value="LastNameIndex">
        <extension vendor-name="FastObjects" key="unique"
                    value="false"/>
        <extension vendor-name="FastObjects" key="member"
                    value="lastName"/>
    </extension>
</class>
```

2. Select the index in the application code, using one of the following `com.poet.jdo.Extents.iterator()` methods:

```
Iterator iterator(Extent extent,
                  String indexName,
                  boolean ascending);

Iterator iterator(Extent extent,
                  String indexName);
```

The first parameter specifies an `Extent` object, the second the name of the index. This is the name defined in the database schema. Make sure that you use the exact index name specified in the schema. The `iterator()` method will throw an `Exception` if this index name is not found. Indexes of `superclasses` cannot be used.

The third parameter provides the sort order. The sort order may be ascending or descending. The default is ascending.

In the following example, the `Extent.iterator()` method is used to retrieve objects from the `Student` extent in name sorted order.

```
Extent students = pm.getExtent(Student.class,true);
Student student;
// sort by LastName index in ascending order
java.util.Iterator iter = com.poet.jdo.Extents.iterator(students,
"LastNameIndex");
while(iter.hasNext())
{ // assuming you have defined a toString() method for Student
      System.out.println(iter.next());
}
students.close(iter);
```

7.2 Finding Objects by Index Keys

If speed is essential, you can search for objects using an indexed search. Indexed searches look at the data in the index associated with an `Extent`. Different search methods of the `com.poet.jdo.Extents` class allow you to perform indexed searches for objects based on their indexed values. These methods are almost always much faster than using queries since there is less overhead when compared to compiling and optimizing the query.

7.2.1 Finding Objects by `selectKey`

If, for instance, an application needs to find a `Person` object by `lastName`, an indexed search is the fastest way to find that object, assuming the `Person` object is indexed by its `lastName` data-member.

The indexed values of an object are the values that make up a specific index that is defined for a class. For example, the `Person` class has an index named `LastNameIndex`. This index contains the data-member `lastName`:

Following is the index definition in the `Person` JDO metadata file:

```
<class name="Person">
    <extension vendor-name="FastObjects" key="index"
                value="LastNameIndex">
        <extension vendor-name="FastObjects" key="unique"
                    value="false"/>
        <extension vendor-name="FastObjects" key="member"
                    value="lastName"/>
    </extension>
</class>
```

You can then search for `Person` objects by name using this index and the `Extents.selectKey()` method by specifying the value of the `lastName` member as the search criterion. The `selectKey()` method works directly with indexes. The `selectKey()` method can only be run

on an `Extent Iterator`, for which an appropriate index has been selected.

```
// select an index before searching
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,"LastNameIndex");
// get an Iterator with all Persons matching the specified lastName
iter = com.poet.jdo.Extents.selectKey(iter,"Miller",true);
while (iter.hasNext())
{
// traverse all matching objects
    Person next = (Person) iter.next();
}
```

Instead of `selectKey` you can also use the following `Extent Iterator`:

```
// get an Iterator with all Persons matching the specified lastName
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,
                                  "LastNameIndex","Miller",true);
```

7.2.2 Finding Objects by `findKey`

Another possibility is to use the `findKey` function, which searches for the first occurrence of a member that matches the key. It positions the `Extent Iterator` on the first object with the specified key. The `findKey` method can only be run on an `Extent Iterator` for which an appropriate index has been selected. You can use this option if you want to iterate all objects with a key value greater than a specified value.

In our School Enrollment Example assume you want to print a list of all persons whose last name is "Miller" or after "Miller" in alphabetical order.

```
// select an index before searching
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,"LastNameIndex");
// position the Iterator on first matching object
boolean found = com.poet.jdo.Extents.findKey(iter,"Miller");
if (found)
{
// get the first matching object
    Person first = (Person) com.poet.jdo.Extents.current(iter);
```

```
    while (iter.hasNext())
    {
// traverse all subsequent objects with key >= specified key
        Person next = (Person) iter.next();
    }
}
```

If an object matching the index key exists, `true` is returned. If the key value is not present, the `Iterator` is positioned on the first object with a key value greater than the searched for key value and the return value is `false`.

7.2.3 Finding Objects by Using `selectRange`

If you are searching for objects based on data that spans a defined range, then the `selectRange` method of `com.poet.jdo.Extents` can be used. Using `selectRange` is significantly faster than a query for finding objects whose data falls within a range of values. This method restricts the `Extent Iterator` to the range specified. You can think of the `selectRange` method as a `selectKey` but with both upper and lower bounds. Following the call to `selectRange`, as you walk the `Iterator`, only the objects whose data falls within the specified range will be traversed.

As with the `selectKey` method, `selectRange` can only be used with an index. Here is an example call to `selectRange` that will limit the `Extent Iterator` to those `Person` objects whose `lastName` is in the range “H” through “K”.

```
// select an index before searching
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,"LastNameIndex");

// get an Iterator with all Persons matching the specified lastName
iter = com.poet.jdo.Extents.selectRange(iter,"H","L",true,false,true);
```

These are the parameters of the `selectRange()` method:

```
java.util.Iterator selectRange(java.util.Iterator iter,
                               java.lang.Object lowerBound,
                               java.lang.Object upperBound,
                               boolean lowerInclusive,
                               boolean upperInclusive,
                               boolean forwards)
```

The way in which index search keys must be specified was described in the description of the `selectKey` method, above. Here you will need two search keys: one to specify the value of the lower bound and one to specify the upper bound. For our example call above, the `Extent Iterator` will be positioned at the object whose `lastName` has the value “H”.

The `lowerInclusive` parameter specifies whether the lower bound is included in the range or whether the range starts with the next “greater” value. I.e., is the specified value included in the range or not. The same applies to the `upperInclusive` parameter. It specifies whether to include objects with the specified upper bound or stop just before objects with the upper bound value. In the example, we want all objects with `lastName` in the range “H” through “K”. So, we specified a lower bound of “H” and set `lowerInclusive` to `true` (just in case someone has the last name “H”). The upper bound is set to “L” (the letter following “K”) and `upperInclusive` to `false`. A `lastName` of “Kzzzzzz” will be in the range but “L” will not.

The last parameter specifies the traversal direction. Setting `forwards` to `false` will traverse the range in reversed direction. This allows you to decide between ascending and descending order.

Instead of `selectRange()` you can also use the following `Extent Iterator`:

```
// get an Iterator with all Persons matching the specified lastName
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,
                                  "LastNameIndex","H","K",true,false,true);
```

7.2.4 Index Key Specifications

In the example above, `LastNameIndex` is an index on a `String` field. Therefore, a `String` value has to be passed to the `selectKey` call. If the indexed field is of a primitive type (e.g. `int`), you have to pass an instance of the corresponding wrapper type (e.g. `java.lang.Integer`).

If the index is a compound index, an `Object[]` array filled with objects of the appropriate types of the indexed fields has to be passed. The following example shows the usage of compound indexes.

In our School Enrollment Example, assume you want to find all persons with a specific last name and birthday.

Compound index definition in the `Person` JDO metadata file:

```
<class name="Person">
    <extension vendor-name="FastObjects" key="index"
               value="NameBirthdayIndex">
        <extension vendor-name="FastObjects" key="unique"
                   value="false"/>
        <extension vendor-name="FastObjects" key="member"
                   value="name"/>
        <extension vendor-name="FastObjects" key="member"
                   value="birthday"/>
    </extension>
</class>
```

Searches can now be done in the following way:

```
// select an index before searching
Extent persons = pm.getExtent(Person.class,true);
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(persons,"NameBirthdayIndex");

// get an Iterator with all Persons
// named "Miller" with birthday on 02/17/1965
Object[] key = {new String("Miller"),new String("02/17/1965")};
iter = com.poet.jdo.Extents.selectKey(iter,key,true);
```

7.2.5 Limitations

The search based on index keys has the following limitations:

- No wildcards are supported for strings.
- Searching beyond the defined significance of a string is not possible. For instance, the `LastNameIndex` index has a significance of ten by default, because a value was not specified. Only the first ten characters of the name are stored in the `LastNameIndex` index. In this case, only searching based on the first ten characters of the data-member `lastName` is possible.

7.3 Finding Objects by Using Filtered Extents

Sometimes it is necessary to iterate through only those instances of a certain class that fulfill a certain criterion. One way to achieve this is to collect the desired elements into a collection by execution of a query. In this section we present a different approach, namely filtered extents. A filter is an OQL predicate `String` with a free variable '`this`'.

In contradiction to `selectKey, findKey` and `selectRange`, filtered `Extent Iterators` can be used without specifying an index. Filters on `Extent Iterators` are very similar to queries, but they fetch results one (or `n+1` if you use `setPreFetch(iter,n)` for that `Extent`) at a time, also in client/server environment—a difference that can be very important when you write browsers or share a server on a network. For instance, suppose a user performs a query that returns 1000 items.

If you display the results in a window that holds 50 items, you may want to display each item as soon as it is found, and you would like to stop as soon as the window is full. If the user scrolls to the next page, you could then get the next 50 items.

Queries		**Filters**	
find item 1		find item 1	
find item 2			display item 1
...		find item 2	
find item 1000			display item 2
	display item 1	...	
	display item 2		...
	...	find item 1000	
	display item 1000		display item 1000

By design, queries always find all results. This means that you cannot start displaying the initial items that are found until all items have been found. You can circumvent this limitation by using filters, which allow you to display each item as soon as it is found.

7.3.1 The `setFilter` Method

A filter is simply a query specification that "modifies" an `Extent Iterator`. It is set with the `com.poet.jdo.Extents.setFilter` method. The filter critreria are specified in `OQL` syntax. Sorting arguments are not allowed within the query expression. In contradiction to `selectKey`, `findKey` and `selectRange`, filters can be used without specifying an index.

When you retrieve an item from a filtered `Extent Iterator`, you see only the items that satisfy the query conditions that you specified in the filter. In other words, the `Iterator` acts as though it were the result collection for the query, but instead of performing the query all at once, FastObjects finds each item as you read it from the filtered `Extent`.

The following is a simple example that lists the persons in a database that satisfy the conditions set by a given filter. The filter is defined to find all persons which last names begin with "F".

```
// assuming pm represents the current PersistenceManager
Extent personExtent = pm.getExtent(Person.class,true);
java.util.Iterator iter = personExtent.iterator();
com.poet.jdo.Extents.setFilter(iter,
             "WHERE this.lastName LIKE \"F*\"");
```

You may also specify a read ahead.

```
com.poet.jdo.Extents.setPreFetch(iter,10);
```

Now you can read all elements out of the filtered collection—only the persons who match the query specification will be found:

```
while (iter.hasNext())
      {
    Person person = (Person) iter.next();
// assuming you have defined a toString() method for Person
    System.out.println(person);
    pm.evict(person);
}
```

`PersistenceManager.evict` is called to immediately release resources held by the `Person` object.

7.3.2 Sort Orders for Filtered Extent Iterators

When indexes are available, the default behavior is that the filter uses the best index, not the current index order of the `Extent Iterator`. The elements are returned in the order of the best index. If you want to use the current index, you need to set the `useCurrentIndexForFilter` transaction property to `true`. In this case the elements are returned in the order of the current index, but this may decrease the performance of the filter.

For instance, the following example demonstrates a search for people whose last name starts with “F”, sorted by the last name in descending order:

```
Transaction txn = pm.currentTransaction();
java.util.Properties p = com.poet.jdo.Transactions.getProperties(txn);
p.put("useCurrentIndexForFilter", "true");
com.poet.jdo.Transactions.setProperties(txn, p);
txn.begin();

Extent personExtent = pm.getExtent(Person.class,true);

// use an indexed Extent Iterator with descending sort order
java.util.Iterator iter =
    com.poet.jdo.Extents.iterator(personExtent,"LastNameIndex",false);
com.poet.jdo.Extents.setFilter(iter,"WHERE this.lastName LIKE \"F*\"");

while (iter.hasNext())
{
    Person person = (Person) iter.next();
// assuming you have defined a toString() method for Person
    System.out.println(person);
    pm.evict(person);
}
```

Keep in mind, however, that specifying a sort order may slow down the filter considerably if you force FastObjects to use an index that has no relationship to the filter. The preceding example is fine because the filter requires FastObjects to look through the `lastName` members, and the

`LastNameIndex` is a reasonable way to do this. FastObjects simply finds the first person whose `lastName` begins with "F" and walks the `LastNameIndex` until it finds someone whose name does not begin with "F". If, however, if you want FastObjects to find all people whose first name (`firstName` member) starts with "'F", using the specified index on the `lastName` members, then FastObjects would have to search the entire `LastNameIndex` sequentially, returning a value every time it found someone whose `firstName` starts with "F", which of course slows down performance.

7.3.3 Filter Limitations

Filters are better than queries for many tasks, but they do have some limitations that you need to be aware of:

- Filters are currently available only for `Extents`, so you will have to do queries to examine other kinds of collections.
- Filters do not nest. An `Extent Iterator` may have only one filter at a given time.
- Queries on filtered `Extent Iterators` currently ignore the filter entirely and act as though the filter had never been set.
- The query that is applied as a filter must always return a collection of persistent objects. Queries that return single objects, base types, or collections of base types are not allowed. For instance, if you use the `COUNT` statement, an integer is returned as the result of this query. This makes no sense when applying it as a filter, as the filtered `Extent Iterator` has to return object references.
- `ORDER BY` clauses are not allowed.

7.4 Finding Objects by Using Queries

The FastObjects JDO Binding provides two different query languages: JDOQL and OQL.

JDOQL is the JDO query language, defined in the JDO specification. Its syntax follows the Java syntax. JDOQL is used to formulate the query filter string required by the JDO `Query` object. The `Query` object applies the query filter to a so-called candidate collection, which may be a collection or an extent, and returns all matching objects from the candidate collection to a unmodifiable `Collection`. Please refer to the JDO specification for more information on JDOQL.

Example:

```
// assuming pm the current PersistenceManager
Extent persons = pm.getExtent(Person.class,true);
String filter = "lastName == \"Miller\"";
Query jdoqlQuery = pm.newQuery(persons,filter);
Collection result = (Collection) jdoqlQuery.execute();
// result contains Person objects with last name="Miller"
...
jdoqlQuery.close(result);
```

OQL, the Object Query Language, defined by the Object Data Management Group (ODMG) is a standard query language for object database management systems. It is a declarative query language based on the syntax of Structured Query Language (SQL), the query language used with relational databases. The basic querying construct is the same as that used in SQL, that is, a `SELECT...FROM...WHERE` statement. OQL statements are simply text strings. For detailed information on OQL please refer to the *FastObjects OQL Reference*.

In FastObjects, an OQL query can be directly executed in the JDO Binding by creating a `Query` object using the `newQuery(String language, Object query)` method provided by the `PersistenceManager`. The `language` parameter specifies the query language. For an OQL query this parameter must be "`org.odmg.OQL`". `The query` parameter is the query instance. The required class of this instance is defined by the specified query language. For OQL it must be `String`. The result of such an OQL query is an unmodifiable `Collection` as for JDOQL.

Example:

```
String query = "select persons from PersonExtent AS persons "
                + " where persons.lastName = \"Miller\"";
// assuming pm the current PersistenceManager
Query oqlQuery = pm.newQuery("org.odmg.OQL",query);
Collection result = (Collection) oqlQuery.execute();
// result contains Person objects with last name="Miller"
...
oqlQuery.close(result);
```

In the following sections we will only concentrate on the filter string for JDOQL or OQL, respectively. Creating and executing of a query will be the same as above.

7.4.1 Execution Time of Queries

The time a query needs for execution mainly depends on:

- The number of objects that must be inspected
- The number of results comprising the final and all intermediate results that may be generated by the FastObjects Query Optimizer
- The way in which indexes are used by the FastObjects Query Optimizer

7.4.2 Building Optimal Search Phrases for the FastObjects Query Optimizer

The FastObjects Query Optimizer is a software component that optimizes query and filter requests. The optimizer examines the query request, consults information it holds on the content and structure of the database, and determines the fastest way to execute the query request.

This is not a simple task. Because of the variety and complexity of object relationships in the database, there may be a number of ways to execute

a query, one of which will be the fastest. It is the job of the optimizer to identify the fastest method available and execute it.

The optimizer often works with indexes, automatically selecting and using indexes during complex queries. Since the query optimizer works automatically with queries, you do not have to worry about selecting indexes when performing a query. However, if a query requires sorting on `lastName`, but an index is not defined for `lastName`, then the optimizer must resort to sorting the result collection.

The query optimizer generates a query tree (`QTree`) and a query plan (`QPlan`).

The main job of a query tree is to perform the comparison on buffer content, index keys or memory objects and also hold intermediate results. It builds a new result starting with several intermediate results depending on the logical operation (conjunction/disjunction).

The query tree consists of one or more query tree nodes. Every query tree node represents a subquery, a sorting instruction or a logical operation. Each query tree node has a type:

- type member condition (`TYPE`)
- structure component like subclass (`SUBCLASS`)
- logical operation (`AND,OR`)
- sortby criteria (`SORT`)
- logical constant `true` or `false`.

The query plan defines the way in which the query tree is executed. It defines which of the query tree nodes is to be executed first, when and how results of query tree nodes have to be combined, etc.

The query plan is a sorted list of possible query plan tasks (`QPTask`); tasks that depend on another task are executed in order. If no indexes are available, the query plan only consists of one query plan task, the main task. The main task represents the query tree. If indexes are available, the query plan can consist of more than one query plan task. Every query plan task then represents a sub-query tree and one of the query plan tasks represents the query tree as a whole (the main task). In general, a query plan task encloses all conditions of one sub-query tree within the object network and is also responsible to generate a result for its sub-query tree.

A query plan task is a sorted list of query plan nodes (`QPNode`). For every condition that may be resolved via index access, one query plan node is built. This means that each query plan node consists of just one query tree node of type `TYPE` or `SUBCLASS` and `AND/OR` conditions using the same index. Before executing the next node, and if the logical operation between the previous node and the current is a conjunction, the result of the previous node will be set as a source list for the current node. In that case, the usefulness of the execution of the node will be checked. If the execution of the current node seems to be more expensive than buffer comparison, the node will not be executed. If the node is executed, every matching object will be checked against the source list coming from the previous node and only the objects also matching the source list will be kept in the result set. After all nodes are executed, the results are combined as the result set of the query plan task.

Every query plan node will be executed in the following manner:

If there is an index that can be used, the appropriate sub-query tree and information on boundaries are passed to the index service. If the optimizer could not find an index that can be used, the query plan node uses the extent of the class without any boundary information.

The query plan node then works as follows:

1. If there is a lower bound, the query tree node is positioned to that lower bound, otherwise to the first entry of the index or extent.
2. The query tree node loops through the index/extent until the end of the index/extent or the upper bound (if given) is reached. Within the loop every entry will be checked against:

- Existence within a source list (if given). The source list is the result set of a previously executed query plan node.
- Correct scope, which means it is checked that the object is of the correct class. To illustrate this, consider students are searched by their last name and no `lastNameIndex` is defined on the `Student` class but on the `Person` class. In this case the query optimizer would take this index. But this index contains all students and also all instructors. So for every object found, it must be checked that it is really a `Student` object.
- The query condition (compare). If the query plan node is executed on an extent, the comparison will be done directly on the previously read object buffer.

3. If all succeeds, the entry will be put into the result set together with additional information such as sortby values and compare warnings (due to truncated strings).

At last, the query plan task checks if there are unresolved issues such as compare warnings (resulting of truncated strings due to a too short chosen index significance) or conditions which are not resolved via index query. If so, the task iterates over the given result set and resolves those issues by comparing the disk buffer. If the task is the main task it builds the final (sorted) result set.

7.4.2.1 The Query Optimizer with Search Criteria (Filter and WHERE Clauses)

A common area for optimization is the filter expression of a query. Following is a simple example for a query on the `Person Extent` to find all persons whose last name begins with "Q" or greater:

The JDOQL filter string will look like this:

```
lastName > "Q"
```

The OQL query string will look like this:

```
SELECT p FROM PersonExtent as p WHERE p.lastName > "Q"
```

Assume we have defined an index `LastNameIndex` on the `lastName` data-member of the `Person` class. The query optimizer can either read all `Person` objects sequentially and later disregard those that do not qualify, or use the index and select all those beginning with "Q". It knows that the `LastNameIndex` index will quickly find all `Person` objects with

`lastName` starting with "Q". But what about the rest of the objects? Do they make up half of the database or less?

The best choice depends on what proportion of the objects has names beginning with "Q". If the proportion is small, it will probably be best to use the index, but if the proportion is high, it will be faster to read the whole `Extent`. To help determine the optimal solution, the optimizer uses statistics on the distribution of indexed values to aid in making the best choice.

Another point to make here that can help the query run faster. Use < and > where possible instead of <= and >= (for example "`age < 10`" instead of "`age <= 9`"). This will eliminate a second test on a data-member.

7.4.2.2 The Query Optimizer with Query Ordering

The ordering of a JDOQL query (using `setOrdering(String ordering)`) or of an OQL query (using the `ORDER BY` clause) may cause a query to be sorted. The optimizer must determine if extra sorting is necessary based on the defined and used indexes.

The current index defined for an `Extent` is ignored during queries when using ordering.

When using indexed string-values, the optimizer is capable of sorting beyond the significance of the index. This can be demonstrated using the previous example of the `Person` class, where the index defined on the data-member `lastName` was called `LastNameIndex`. Since a significance for this index was not specified, by default, only the first 10 characters of the string were stored in the index. Even though the index only has a 10-character significance, the optimizer is still capable of looking into the objects and sorting to the full extent of the name string.

7.4.2.3 Grouping Expressions to Give Hints to the FastObjects Query Optimizer

If you execute queries that include data ranges, you should group expressions in the following way:

```
JDOQL:      gender == 'm'
      && (lastName >= "H" && lastName < "K")
OQL:  SELECT * FROM PersonExtent AS x
      WHERE x.gender = 'm'
      AND (x.lastName >= "H"
           AND x.lastName < "K");
```

This helps the query tree builder to group the range criteria together. It supports the query optimizer in finding the optimal ranges for an index or at least to find it faster.

7.4.2.4 Not Using Unsupported Term Transformations

The query optimizer does not support term transformation according to the following rule:

`a AND (b OR c)` ←→ `a AND b OR a AND c`

Suppose, for example, that you have defined a compound index on `lastName` + `firstName` members of the `Person` class and you want to find all persons named "Miller" and with a first name of "Tom" or "Jerry".

If you use the following query it will not make efficient usage of the compound index:

```
JDOQL:      (lastName == "Miller")
      && (firstName == "Tom"
          || firstName == "Jerry")
OQL:  SELECT * FROM PersonExtent AS x
      WHERE x.lastName = "Miller"
      AND (x.firstName = "Tom"
           OR x.firstName = "Jerry");
```

You would be better off using the following syntax:

```
JDOQL:      (lastName == "Miller" && firstName ==
"Tom")
      || (lastName == "Miller"
          && firstName == "Jerry")
OQL:  SELECT * FROM PersonExtent AS x
      WHERE (x.lastName = "Miller"
             AND x.firstName = "Tom")
      OR (x.lastName = "Miller"
          AND x.firstName = "Jerry");
```

7.4.2.5 Avoiding Projections With Large Result Collections in OQL Queries

This is only an issue for OQL, because JDOQL does not support projections.

In our School Enrollment example, suppose you want to print a list of the names of all instructors holding courses in a specific room, e.g. the room named "A-1001".

You can formulate an OQL query like this, containing a projection:

```
OQL:  SELECT c.instructor.lastName
      FROM CourseExtent AS c
      WHERE c.room.name="A-1001"
```

You may think this a good solution, saving time and memory. After all, you do not want to retrieve the complete `Course` object for matches but only the name of the course instructor (perhaps to print it to a list). But at present, in client/server environments, FastObjects executes projections on the client side. So, what really happens is that all matching objects are collected in a query result collection and this collection is sent to the client. The client needs to perform the projection and therefore requests all objects in the collection from the server in order to extract the names.

For result collections with a small number of objects, the performance may be entirely acceptable. But the more objects a result collection contains, the more the performance will affected and you should think about another way to retrieve the needed members.

The above expression would cause all matching `Course` objects to be loaded to the client and then loading all referenced `Instructor` objects. It would be much better to express the query using the `Instructor Extent`:

OQL:
```
SELECT i.lastName FROM InstructorExtent
AS i,i.courseSet AS course
WHERE ((Course)course).room.name = "A-1001"
```

Using this expression avoids the loading of the matching `Course` objects to the client. Only the matching `Instructor` objects would be loaded. These objects are needed anyway to print the list of names.

In JDOQL you can express the filter like this:

```
// assuming pm the current PersistenceManager
Extent instructors = pm.getExtent(Instructor.class,true);
String filter = "courseSet.contains(r) && r.name == \"A-1001\"";
Query jdoqlQuery = pm.newQuery(instructors,filter);
jdoqlQuery.declareVariables("Room r");
Collection result = (Collection) jdoqlQuery.execute();
// result contains all Instructor objects with at
// least one course held in room A-1001
// to print the instructor names you must iterate
// through the result collection
...
jdoqlQuery.close(result);
```

7.4.3 Tracing the Query Execution

Because sometimes it is not clear why a query is executed differently than expected, the query execution can be configured to provide more output. This is for debugging purposes only and can be used to see how many objects are processed in each stage of the query, how the query is optimized, and which indexes are used. To configure the query to print out more information, you need to provide the following FastObjects configuration file entries before FastObjects is loaded (the usual name of the FastObjects configuration file is `poet.cfg`):

```
[debug]
POETConsole=3; 1 is for stderr, 2 is for OutputDebugString
query=4
```

Continuing with the School Enrollment example, the use of an index with query trace is demonstrated. Assume you perform a query on `Person` objects:

JDOQL:
```
lastName >= "D"
```

where an index `LastNameIndex` is available. A sample output can then be as follows:

The part following `QueryTree:` contains the query string as seen by the query optimizer. After the optimizer has processed it, the (transformed) query expression is printed out again (see part `optimized Tree:`). Note the `"I:"` after the `TYPE` statement. This indicates that this particular condition can use an index. The index name is later printed out in the query plan `QPlan` part. The very last part of the output shows the quantities of objects compared within the index and the number of matches, as well as the bounds to be used for that index.

```
// Build and optimize query tree and query plan
// Query as seen by the query optimizer
QueryTree:
TYPE (val=0){'lastName' >= "D"};

// Query as optimized by the query optimizer
// the value of the parameter val printed in parentheses is the rating
// computed by the query optimizer. It is the assumed number of objects
// to be found. The "I:" indicates that an index is available
optimized Tree:
TYPE (I:val=10.8408->QNode-2){'lastName' >= "D"};

// Query plan
// the following line summarizes the number of tasks that were set up,
// the number of tasks that will be analyzed (valid tasks)
// and the number of sorts that will be done in the query plan
QPlan: with 1 tasks, 1 valid tasks, 0 sorts
// Name of the query plan task and class to work on
 [0] QPTask-1 on class com.poet.schoolenrollement.Person
          with 1 nodes, 1 valid nodes, 0 sorts
// Sub-query tree the query plan task executes
        QTree:
          TYPE (I:val=10.8408->QNode-2){'lastName' >= "D"};
// Name of the query plan node(s) the query plan tasks consists of,
// name of the index to use and class to work on
    [0] QNode-2: uses index LastNameIndex of class
                    com.poet.schoolenrollment.Person
// Sub-query tree to be executed by query plan node
                QTree:
          TYPE (I:val=10.8408->QNode-2){'lastName' >= "D"};

//Execution of query plan
{--------------------------------------->> QPTask-1 starting:
(.....................................>> QPNode-2 starting:
// Sub-query tree to be executed by query plan node
QTree:
```

```
TYPE (I:val=10.8408->QNode-2){'lastName' >= "D"};

// Numbers of boundaries and indexes to be used by the query plan node
BoundsNum: 1  Query uses Index :00102_LastNameIndex

// Start with first (here the only one) boundary
Precondition of boundary 0:
// Query tree node to be executed
TYPE (I:val=10.8408->QNode-2){'lastName' >= "D"};

//Execution statistics of query plan node
// The query operates in the range from >= D (lower bound)
// to end of index (upper bound).
// This range contains 10 entries of 13 entries
// in the complete index (checked).
// source list is NO because no other query tree nodes were executed before
MIMService::Query statistic : err = 0
    checked    : 10 of 13
    num of hits: 10 of 10 compares
    .. of warns: 0
    lower bound: >= D
    upper bound: <nil>
source list: NO with 0 entries
MIMService::Query found 10 results
QPNode-2 completed in 0.10s, err=0
<<.....................................)
QPTask-1 completed in 0.10s, err=0
<<-------------------------------------}
Query found 10 results in 0.20s
```

7.4.4 Using Indexes for Effective Queries

Value-based queries can be very slow, especially if they have to examine every record in the database. Indexes can dramatically speed up queries without changing the way they are programmed.

The FastObjects query optimizer searches for a usable index for every query attribute. If the current class does not have such an index, the base classes are inspected. This implies that it is not necessary to have a member indexed in every derived class. Sometimes it might be useful also to add such an index to a derived class, particularly if the base class has significant more objects than the derived class and the query on the derived class is performance critical.

If the query consists of a boolean expression on different attributes, the query optimizer tries to find the best usable compound index. The best compound index is the one that contains the greatest number of the queried attributes and has the fewest unnecessary attributes.

7.4.5 Using Compound Indexes

If you often use queries that combine searches on several attributes, defining a compound index may be more powerful than using several single indexes. For example, if the member `lastName` is sometimes combined with `firstName` and sometimes combined with `birthday`, a compound index on the members `lastName` + `firstName` + `birthday` or `lastName` + `birthday` + `firstName` may be useful. A query can also make use of only the first component of a compound index. If only the first component of the index is used in the query, the other components of the index may be omitted. If you design a compound index, you should order the components by their significance to the query.

7.4.6 Using Indexes for Effective Queries on Sub-Object Attributes

Referring to the earlier example of a query to return courses taking place in a particular room named "A-1001" (refer to section Using Backward References Instead of Queries). The relationship there was modeled programmatically by maintaining a collection of backward references from the `Room` to the `Course` class instead of using a query on the `Course Extent`. Using a query on the `name` attribute of the `Room` sub-object together with an index on object identity eliminates the need for backward pointers and modification of two objects. The tradeoff is that it requires an index to be built and maintained.

Queries on attributes of sub-objects are dramatically sped up when defining indexes (separate in each class) for the reference to a sub-object (i.e. an index on object identity) and for the member of a sub-object. The following query is sped up by indices on `course.room` and `room.name`:

JDOQL:
```
room.name == "A-1001"
```

OQL:
```
SELECT * from CourseExtent AS course
WHERE course.room.name = "A-1001"
```

Index definition in the JDO metadata files:

```
<class name="Course">
    <extension vendor-name="FastObjects" key="index"
               value="RoomIndex">
        <extension vendor-name="FastObjects" key="unique"
                   value="false"/>
        <extension vendor-name="FastObjects" key="member"
                   value="room"/>
    </extension>
</class>
```

```
<class name="Room">
    <extension vendor-name="FastObjects" key="index"
                value="NameIndex">
        <extension vendor-name="FastObjects" key="unique"
                    value="false"/>
        <extension vendor-name="FastObjects" key="member"
                    value="name"/>
    </extension>
</class>
```

Also refer to the section Using Backward References With Indexes Instead of Large or Changing Collections for another example of using an index on object identity and a query instead of defining a collection member.

7.4.7 Considering Query Performance vs. Update Performance When Defining Indexes

Indexes will improve access time but slow down storage time. Therefore, it is critical that the application developer understands the manner in which objects will be accessed and stored and to judiciously select where to use indexes. The reason storage time increases is because FastObjects updates a corresponding index tree for each index in the correspoding class when an object is added, updated, or deleted.

Of course, if you are using the FastObjects special features `findKey`, `selectKey` or `selectRange`, which are implemented in the class `com.poet.jdo.Extents` (and in `com.poet.odmg.Extent`), you will need to define an index for the appropriate class data-member(s). By taking advantage of FastObjects schema evolution functionality, indexes can be fine-tuned even after an application is well into development or even after deployment.

7.4.8 Choosing an Appropriate Index Significance

For indexes defined for a `String` member, the significance might also effect the query performance. If the query uses an index whose significance is too short, the query has to examine the record in the database. For example, the index on `lastName` is set to 3 and the database contains the objects "Baker","Butcher","Miller","Millner","Millery","Military" and "Miltner". The query:

JDOQL:
```
lastName == "Miller"
```

OQL:
```
SELECT * FROM PersonExtent AS x
WHERE x.lastName = "Miller";
```

will examine five objects on disk because in the index, these five objects are referenced by the "Mil" key. But be careful by increasing the significance, higher significance results in larger index sizes and this decreases store/delete performance.