

WHITE PAPER

# OBJECTS, DATABASES AND THE MYTH OF SERIALIZATION

By Robert Greene,  
Vice President Open Source Operations, Versant Corp.

Sponsored by Versant Corporation

## Challenge of Complexity

More complex applications require more complex data.

Serialization therefore has become a common practice for storing object graphs – but it comes at a cost.

Object databases avoid serialization, are highly efficient and perform much faster than other databases.

---

## INTRODUCTION

Given the complexity of today's data systems, articles continue to be written offering solutions to data management challenges using objects and object databases. Such papers typically focus on the ways in which object graphs<sup>1</sup> are stored and made available to the application space. Put simply, object graphs provide the application with a means to store and retrieve object instances and a structure within which to save the state of its objects at a particular moment in time.

It stands to reason that more complex applications will require more complex object graphs. These object graphs, unlike flat data structures, do not fit well into the tables, rows and columns of a relational database management system. Therefore, serialization—the process of converting an object into bits for storage in a file—has become a common practice for storing object graphs alongside simple, structured data in a database. The result of the serialization process, often stored in a relational database, is what has come to be known as a Binary Large Object, or BLOB.

The serialization process required with blob creation comes at a cost. A high cost in development time, for the creation of blob meta data management code, a performance cost as the serialization process is slow and finally with a high dollar cost as CPU power is heavily consumed at runtime. Serialization is also needed for other types of databases, most notably those employing a key-value storage paradigm (a.k.a. key space) and column or field database techniques.

Authors quite commonly—and incorrectly—refer to the process of serialization as part of what is happening when an object database stores an object or object graph. This white paper debunks that misperception, and sets the record straight about the efficiency of

---

<sup>1</sup> An Object graph encompasses a part of or is a complete view of an object system at a particular point in time. Whereas a normal data model such as a UML Class diagram details the relationships between objects, the object graph details a particular instantiation of their instances. Object-oriented applications contain complex webs of interrelated objects. Objects are linked to each other by one object either owning or containing another object or holding a reference to another object. This web of objects is called an object graph and it is the more abstract structure that can be used in discussing an application's state. (source: Wikipedia)

object databases as brought about by the avoidance of serialization.

MYTHS about object databases:

- » **They necessarily employ serialization**
- » **They have more overhead than other types of databases**

FACTS about object databases:

- » **They avoid serialization and the extra code and CPU cycles required**
- » **They are highly efficient at storing complex graph structures**
- » **They require about 40% less code to implement than other types of databases**
- » **They perform between 1.5 and 30 times faster than other types of databases**

---

## WHAT IS SERIALIZATION?

As defined by Wikipedia, serialization in the context of data storage and transmission is "the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link" for consumption by an application or service.

To perform serialization within a Java program, you'll need to do certain things to the source code of your class files, such as:

- » **Extend your classes to implement the Serializable interface.**
- » **Use an ObjectOutputStream with a FileOutputStream to store**
  - a. - Use `writeObject( obj )` on your stream to store into the file.
- » **Use an ObjectInputStream with a FileInputStream to read objects**
  - a. - Use `readObject( obj )` on your stream to restore ...must cast too
  - b. - Use the `obj` to process business logic

## Serialization – the side effects

Managing large graphs is inefficient.

There is no object versioning available

It's difficult to manage incremental changes.

---

## SOME PITFALLS OF SERIALIZATION

Along with serialization come a few severe side effects, including:

### **Managing small portions of large graphs is quite inefficient.**

All objects referenced by a root object (obj) will also get serialized and written to file along with the root. If there are 100K objects referenced by (obj), then all 100K objs will make it into the file. Likewise, when you read and cast (obj) into your application, all of those 100k objects will get instantiated into your JVM and will look just like they did before serialization's writeObject() call was made. In many use cases you typically only want to use the root object or a portion of the graph, and serialization will cause lots of unnecessary processing.

### **Evolving your object model requires object versioning.**

You must be very careful not to evolve the structure of your classes as this will cause an incompatibility when you try to read old objects. Unless you're implementing some versioning code, don't delete any fields or change a type (for example from int to long) or change class hierarchy, etc.

### **Managing incremental change is difficult.**

Of major concern is the way serialization uses streams and how streams handle identity. You must be careful not to update and store multiple changes to a stream because those changes won't be saved into the underlying file. If you want changes to make it to the file, you must close and reopen the stream with every update of your object. This is manageable, but can be cumbersome and add performance overhead if your application requires numerous updates to your persistent objects.

For more  
information on  
Versant Database  
Engine please visit  
[www.versant.com](http://www.versant.com)

---

## WHAT IS AN OBJECT DATABASE?

Object databases have been around for more than 20 years, and are proven tools for managing complex objects and providing object persistence. Some of the key attributes to an object database are:

### **Seamless integration with object oriented languages.**

Unlike SQL—which encompasses its own database language apart from the programming language—the object database uses the OO programming language as its data definition language (DDL) and data manipulation language (DML). The application objects are the database objects. Query is used for use case based optimization, not as the sole means of accessing and manipulating the underlying data.

### **Object databases manage the persistence of any object graph directly.**

There is no application code needed to manage the connectivity between objects or how they are mapped to the underlying database storage. Object databases use and store object identity directly, bypassing the need for the CPU and memory expensive set based JOIN operations using SQL.

**Object databases exhibit traditional database features**, such as queries, transaction handling, backup and recovery along with advanced features such as distribution and fault tolerance.

## Object Databases

- » will reduce CPU consumption
- » provide faster access to objects
- » enable high concurrency and scalability

---

# KEY ADVANTAGES OF THE OBJECT DATABASE

Object databases avoid serialization for a number of reasons:

**It eliminates CPU cycle consumption.** Serializing objects to/from the data store consumes resources. Object databases can simply memory map objects to/from the data store. Further, if you serialize objects to disk or a database and you subsequently want to find the right one using anything other than a simple key, you must maintain a separate meta data layer. Object Databases manage these semantics for the application developer.

**It provides faster and more efficient access to objects.** Having the semantics of the object graph allows object databases to pinpoint objects and load objects only on demand, while serialization must always load and deserialize the entire object graph. So, in order to avoid serialization's overhead of having to read, write and update entire object graphs to and from the application space, object databases allow you to treat each of the objects in a graph as discrete, only loading/unloading those required for each use case.

**Enable fine grain concurrency and full scalability.** From a database perspective, serialized object graphs are completely opaque, and make even the smallest update to the contained data a time-consuming chore that requires locking and updating the entire graph. This is an unnecessary waste of compute resources and doesn't scale across multiple users. In contrast, object databases provide full access to the objects beneath the root of a graph without the requirement to lock and/or load the entire structure.

---

## PUTTING OBJECT DATABASES TO THE TEST

Comparing disparate technologies such as serialization and object database persistence is never simple. A database typically provides a number of important features, including transactions, concurrency control and query support, whereas the most basic form of serialization simply uses a flat file to persist the data.

For the sake of simplicity, we used a flat file to serialize a test graph. It is easy to imagine using the same serialization code when storing the test graph into a BLOB field of a relational database or into the value field of a key value store. Such databases would add their own overhead to the equation, which in turn would result in an additional performance hit.

In the following examples, we used a Versant Object Database System and a local hard drive to benchmark and contrast serialization and object database performance. Over the next few sections, we document a simple development of an object graph and illustrate how serialization and object databases stack up against each other.

---

### BASIC OBJECT STORAGE

First, we look at the performance of basic serialization compared with object databases. With some simple code, we can show just how taxing serialization can be for your CPUs and the performance of your application. In the following, we use serialization to store 1000 objects to a file and compare that to the time it takes to (1) print those 1000 objects to the console in an IDE and (2) store the 1000 objects into an object database.

Where the SerTest class looks like the following and noting that you DO NOT need to implement Serializable for the object database code, we are just keeping the example consistent by using a single class for both use cases:

The code looks as follows:

#### SerTest

Date: Time  
String: attribute

... methods

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;

public class SerTest implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date time;
    private String attribute;

    public SerTest(){
        time = Calendar.getInstance().getTime();
        attribute = "value";
    }

    public Date getTime() {
        return time;
    }

    public String getAttribute() {
        return attribute;
    }
} // end of class
```

The main program executing the behavior looks like this:

```
package versant.runtime;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import versant.helper.SerTest;

public class SerTestMain {

public static void main(String[] args) {

String filename = "C:\\SerTest.ser";
ObjectOutputStream out = null;
System.out.println("Starting run....");

// SERIALIZATION CODE
try {
    out = new ObjectOutputStream(new
        FileOutputStream(filename));
    long startTime = System.currentTimeMillis();
    for (int i=0; i<1000; i++ )
        out.writeObject(new SerTest());
    out.flush();

System.out.println("Total Time To Serialize to File: " +
    (System.currentTimeMillis()-startTime));

} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    try {
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//CONSOLE OUTPUT CODE
startTime = System.currentTimeMillis();
for (int i = 0; i < 1000; i++) {
    System.out.println(new SerTest());
}

System.out.println("Total Time To Write to Output Console: "
    + (System.currentTimeMillis() - startTime));

//OBJECT DATABASE CODE

//connecting to the database called "sdb"
TransSession session = new TransSession("sdb");

startTime = System.currentTimeMillis();
for (int i = 0; i < 1000; i++) {
    new SerTest();
}
}
```

```
//committing the objects to the database
session.commit();
System.out.println("Total Time To Write to Object Database:
" +(System.currentTimeMillis() - startTime));
} // end of main
} // end of Class
```

The above produces the following output:

```
Starting run.....
Total Time To Serialize to File: 125
versant.helper.SerTest@d16fc1
versant.helper.SerTest@1eb0
....
versant.helper.SerTest@1e55794
versant.helper.SerTest@1d8d237
versant.helper.SerTest@1d13272
versant.helper.SerTest@146e381
Total Time To Write to Output Console: 67
Total Time To Write to Object Database: 62
```

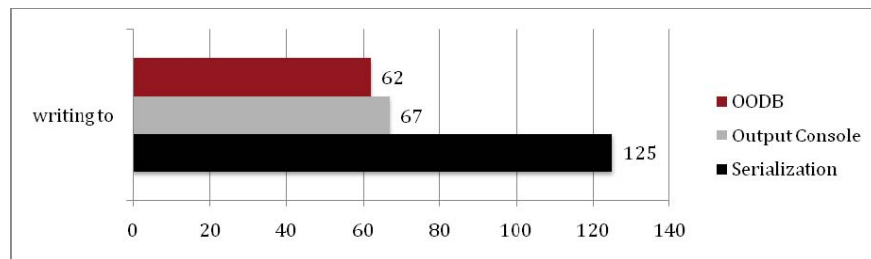


FIGURE 1

Figure 1 shows that Serialization takes twice as long as the other operations. Not shown is that CPU utilization also was correspondingly higher. Of course, this means you are paying the penalty not only in performance but also in cost. Our test was trivial, but as the model and objects become more complex, the differences will be greatly magnified.

## BASIC OBJECT RETRIEVAL

Next we look at object retrieval for basic serialization and for object databases. When using serialization, access to objects involves opening a file and reading all objects as they were written, casting an entry in the file to a reference object and then using the reference. With an object database, you connect to the database and retrieve selected objects: you would either ask for them by name (if bound by name) or perform a query and receive a set based on a criteria.

Of course, forming a query is more complex than simply reading and casting, but it also provides the ability to get at any object without consideration for the storage order, provides for concurrency control in multi-user environments and allows you to retrieve only what you want without having to load reference objects, unless you specifically request. Using a query lets you leverage the metadata present in the database.

Let's look at an example and check out the timing:

```
package com.versant.runtime;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import com.versant.fund.Constants;
import com.versant.fund.QueryExecutionOptions;
import com.versant.internal.helper.SerTest;
import com.versant.trans.Query;
import com.versant.trans.QueryResult;
import com.versant.trans.TransSession;

public class SerTestMainII {

public static void main(String[] args) {

    String filename = "C:\\SerTest.ser";
    ObjectInputStream in = null;
    SerTest ref = null;

    System.out.println("Starting run.....");

    //SERIALIZATION CODE
    long startTime = System.currentTimeMillis();
    try {
        in = new ObjectInputStream(new
            FileInputStream(filename));
        for (int i=0; i<1000; i++){
            ref = (SerTest)in.readObject();
            //Use the object so you know it is in the JVM
            ref.getAttribute();
        }

        System.out.println("Total Time To Read Serialized objects
            from File: " + ( System.currentTimeMillis()-startTime));
    } catch (IOException ex) {
        ex.printStackTrace();
    } catch (ClassNotFoundException ce) {
        System.out.println( "You forgot to run the
            Serialization part first");
    } finally {

        try {
            in.close();
```

```

    } catch (IOException e) {
        e.printStackTrace();
    }
}

//OBJECT DATABASE CODE

Object[] found = null;

//connecting to the database "sdb"
TransSession session = new TransSession("sdb");
startTime = System.currentTimeMillis();

//doing the database query method
found = getSerTestObjs( session );
for( int i=0; i<1000; i++ ){
    ref = (SerTest)found[i];
    //Use the object so you know it is in the JVM
    ref.getAttribute();
}
System.out.println("Total Time To Read Objects from
database: " + ( System.currentTimeMillis()-startTime));
} // end of main

//QUERY CODE

private static Object[] getSerTestObjs( TransSession session
){

//Setup for the fact that you want to return the objects
//To applications space, instead of just references.
QueryExecutionOptions options = new QueryExecutionOptions();
options.setFetchObjects(true);

// Create a query object with criteria ( could have usual
//where clause, logical, aggregation, sort operators, etc )

Query query = new Query( session,
"select * from com.versant.internal.helper.SerTest");

//Apply options and execute the query
query.setQueryExecutionOptions(options);
QueryResult results = query.execute();

return results.next(1000);
}
}

```

The above produces the following output:

```

Starting run.....
Total Time To Read Serialized objects from File: 94
Total Time To Read Objects from Database: 63

```

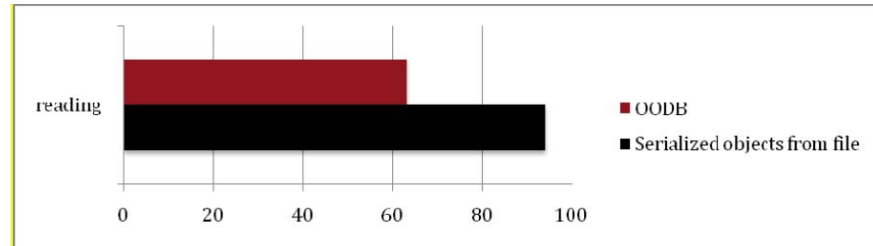


FIGURE 2

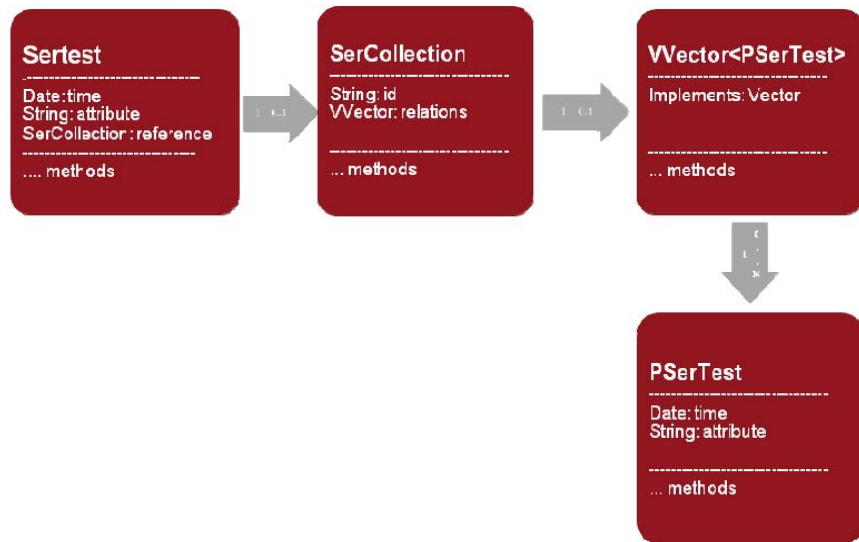
You can see the results in Figure 2.. The absolute numbers are relatively small, but tests consistently show that read speed of the object database is as much as 50% faster. Of course, here we are reading and materializing every object in the database and/or file. The differences would be multiplied if you only wanted to access one of the thousands of objects. We will take a look at this case later in our discussion.

*Note: We do not show how to seek around in a serialized file to get at different serialized objects based on externalized metadata. While qualified selection is very straight forward using the object database and a query, it is completely non-trivial with serialization and as such is beyond the scope of this article.*

## NESTED GRAPH STRUCTURES

So far, we've looked at base performance of the object database compared with serialization. We've seen the powerful metadata and query support which comes with the object database and have seen its superior performance compared to basic retrieval of serialized objects. Now, let's take a look at what happens when you have a deeper set of related objects instead of some flat instance type.

We will modify the SerTest class to include some nested references and check the performance again. We add one additional attribute to the SerTest class called SerCollection. We fill SerCollection with 100 instances of class PSerTest on creation. PSerTest looks exactly like SerTest, except that we used a different class to avoid some recursive instantiation issues.



Here is SerTest with the additional attribute and its setters and getters:

```

package com.versant.internal.helper;

import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;

public class SerTest implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date time;
    private String attribute;

    private SerCollection reference;

    public SerTest() {
        time = Calendar.getInstance().getTime();
        attribute = "value";
        reference = new SerCollection();
    }

    public Date getTime() {
        return time;
    }

    public String getAttribute() {
        return attribute;
    }
} // end of class
  
```

Here is the implementation of the reference type SerCollection:

```
package com.versant.internal.helper;

import java.io.Serializable;
//Notice this collection, it is updateable as a reference
type
import com.versant.util.VVector;

public class SerCollection implements Serializable {

    private static final long serialVersionUID = 1L;
    public String id;
    public VVector relations;

    public SerCollection() {
        id = "cValue";
        relations = new VVector();
        for (int i=0; i<100; i++) {
            PSerTest ref = new PSerTest();
            relations.addElement(ref);
        }
    }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public VVector getRelations() {
        return relations;
    }
    public void addRelation(PSerTest obj) {
        relations.addElement(obj);
    }
} // end of class
```

What is interesting is how the “transparency” materializes in this example extension. In both cases, the runtime code does not need to be modified at all, because when the runtime code instantiates a SerTest instance, it will initialize the referenced SerCollection and all of its contents. Serialization and object databases look similarly transparent when storing deep and nested object graphs. However, that is where the similarities end.

When examining the runtime characteristics, we get the following results:

SerTestMain, the program creating the objects, serializing them to the file and writing them to the object database, provides these results:

```
Starting run.....  
Total Time To Serialize to File: 7063  
Total Time To Write to Database: 7703
```

SerTestMainII, the program that is reading the objects from the serialized file and the object database, provides these results:

```
Starting run.....  
Total Time To Read Serialized objects from File: 2859  
Total Time To Read Objects from Database: 110
```

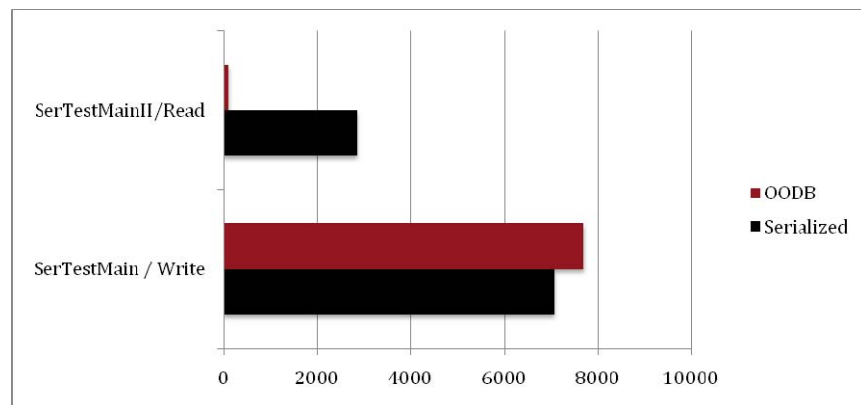


FIGURE 3  
From Figure 3, it is clear that the write performance is basically on par. The object database is only slightly slower, based on creating the necessary meta data infrastructure for new objects and their relationships. The read performance on the other hand, shows several orders of magnitude difference with the object database providing massive improvements over serialization.

Having a little knowledge about what's going on under the covers compels a bit of disclosure. When serialization reads an object into cache, it materializes the entire graph connected to it. However, the object database does not by default load anything other than the objects satisfying the query predicate. It's also possible to only load references. So, in some ways the above is not a fair comparison, except that you have this level of control when using the object database and you have no control when using serialization.

To complete the story, we need to force the object database to load the desired parts or the entire sub-graph of connected objects. Only then can we truly compare the worst case scenario when you actually want the entire connected graph in a use case. This can be accomplished in the object database by adding just one line of code—right after the query gets that first batch of 1000 objects.

The code looks like this:

```
.....found = results.next(1000);  
session.getClosure(found, "sdb", -1, false,  
Constants.RLOCK)
```

When we run again, forcing the load of the entire graph for both serialization and object database, we get the following results:

```
Starting run.....  
Total Time To Read Serialized objects from File: 2812  
Total Time To Read Objects from Database: 1203
```

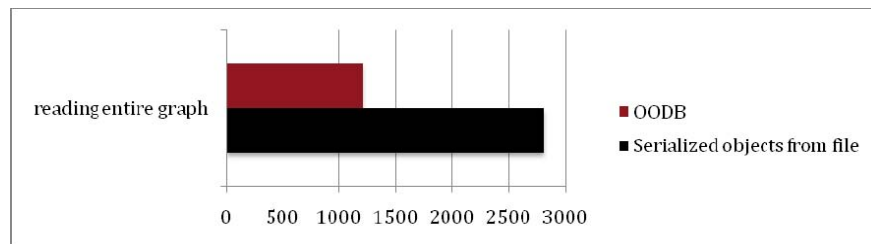


FIGURE 4

You can see the object database is still significantly faster than serialization (Figure 4). As the object model involved gets more complex, e.g., using additional levels of reference, inheritance and recursive relationships, performance differences get wider and a clear advantage emerges in simplicity, performance and scalability for the object database.

---

## OBJECT GRAPH UPDATES

Before we close, let's take a look at a final issue. What happens if we want to create more objects and add those to the SerCollection of an existing SerTest instance? This is considered an update, but it's also a more complex object creation and insertion scenario.

---

### UPDATING AN ENTIRE GRAPH

You can imagine how this goes, first you have to read the existing objects into the application, then iterate those objects and get their corresponding referenced SerCollection and add (in this timing case 100) new objects into the collection. Finally, you need to write them back out to storage in their new form.

This is starting to get repetitive on the coding side, so I will leave the serialization code as an exercise for the reader. In essence, you can take snippets from the above and combine them to do both: the read, then the update and then the output.

Here is the code for the object database to do the update:

```
Object[] found = null;
session = new TransSession("sdb");
startTime = System.currentTimeMillis();

//Query to get the objects just like above
found = getSerTestObjects( session );
for( int i=0; i<1000; i++ ){
    ref = (SerTest)found[i];

    //Change the SerTest's value
    ref.setAttribute("new_value");
    for( int j=0; j<100; j++ ){
        //Add new objects to the collection
        ref.getReference().addRelation(new PSerTest());
    }
}
//Store all changes in an ACID transaction
session.commit();
System.out.println("Total Time To Update Objects from
database: " + ( System.currentTimeMillis()-startTime));
```

Let's look at the new timing. Note we have effectively updated 1000 objects and created 100,000 new objects, in the end having doubled the size of each SerCollection.

```
Starting run.....  
Total Time To Update Serialized objects from File: 34641  
Total Time To Update Objects from Database: 20453
```

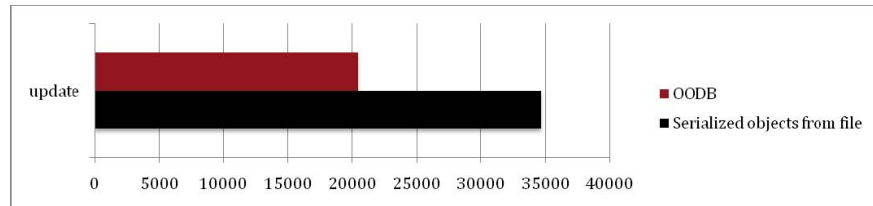


FIGURE 5  
In Figure 5, note that we are literally updating every object in the database and/or file, again with the object database taking far less time than serialization.

---

## UPDATING ONLY ONE OBJECT

What about a scenario where you only wanted to update one object? In the case of serialization, you would still need to read in all connected objects to the one you want to update, do the single update, and then write all the connected objects back out to a file. Plus, you must have some meta data to tell you how to get to that particular object. For example, if you wanted to get at the five hundredth SerTest object directly, you would have to do something like keep track of the size in bytes of each written SerTest, then skip your InputStream the required number of bytes, then read the object and update it. Imagine how hard it would be if your SerTests varied in size. Such a technique would be impractical. The solution would be to read in everything or read sequentially until you get to the one of interest.

However, with an object database, you can simply load the object of interest by value in a query, update it and write that one object back to the database. You do not need to load extraneous objects or keep some kind of customized meta data. Once found and updated, the object will still be fully connected with all its referenced objects.

Clearly, this gives the object database a massively superior performance characteristic over the serialization approach. A really quick hack at such a use case, where we update only the five hundredth instance of the inserted SerTest objects, shows the profile below.

Just remove the loop in the code to get these results:

```
Starting run.....  
Total Time To Update 1 Serialized object from File: 1641  
Total Time To Update 1 Object from Database: 94
```

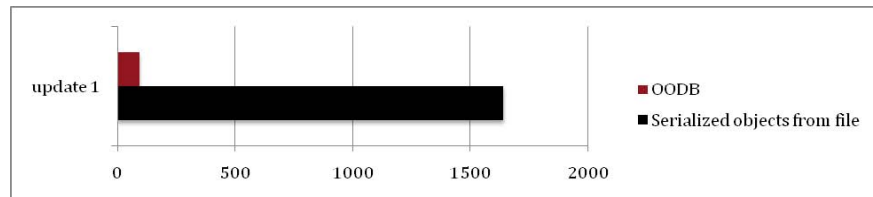


FIGURE 6

---

## CONCLUSION

While the notion of object serialization has often been used to describe the process of storing objects inside a object database, the truth is quite the opposite. Through examples, we've demonstrated the behavior of object serialization and compared it with the behavior of a persistent storage in an object database.

Although from a transparency and ease of use perspective the two approaches might seem similar, the results clearly show that by avoiding serialization, the object database establishes a significantly improved performance profile while at the same time, adds database capabilities, such as transactions, concurrency control, queries, indexing and managing large data sets to the application without adding much to the code complexity.

In addition, the object database has a significantly more efficient data access methodology, and provides a performance profile often exceeding that of serialization by 1 or more orders of magnitude for updates and queries.



### Robert Greene

Vice President,  
Versant Corp.

Robert Greene is  
responsible for Versant's  
open source operations.