A Comparative Study of Persistence Mechanisms for the Java™ Platform

Mick Jordan

A Comparative Study of Persistence Mechanisms for the Java™ Platform

Mick Jordan

SMLI TR-2004-136 September 2004

Abstract:

Access to persistent data is a requirement for the majority of computer applications. The Java programming language and associated run-time environment provide excellent features for the construction of reliable and robust applications, but currently these do not extend to the domain of persistent data. Many mechanisms for managing persistent data have been proposed, some of which are now included in the standard Java platforms, e.g., J2SE™ and J2EE™.

This paper defines a set of criteria by which persistence mechanisms may be compared and then applies the criteria to a representative set of widely used mechanisms. The criteria are evaluated in the context of a widely-known benchmark, which was ported to each of the mechanisms, and include performance and scalability results.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email address:
mick.jordan@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

A Comparative Study of Persistence Mechanisms for the Java™ Platform

Mick Jordan

Sun Microsystems Laboratories

2600 Casey Avenue

Mountain View, CA 94043, USA

1. Introduction

In its relatively short lifetime the Java™ platform has achieved an unprecedented degree of acceptance by industry and academia. The Java programming language synthesized many good ideas from earlier languages and married these with the unassailable heritage of the C family. The simple but pragmatic package mechanism has supported the construction of a large collection of reusable software packages that allows the Java platform to be applied to virtually all segments of the computing landscape.

The language provides a simple, yet powerful, object model, a strong, mostly static, type system, automatic storage management, and concurrency through built-in synchronization mechanisms. These language features are a major factor in the robustness and reliability of applications. However, the Java platform does not extend these virtues to the domain of persistent data (objects). Rather, the Java platform supports the traditional approach, that of input, computation and output, with an explicit transformation to an external representation occurring at the input/output points. In practice, this model is increasingly less applicable, as most applications now require access to volumes of data that demand an incremental approach, for which the transformation code becomes both a source of errors and a time and effort sink.

To mitigate this deficiency, several persistence mechanisms have been developed, many of which are written in the Java programming language. The natural desire to implement the mechanisms in the Java programming language, coupled with a strong resistance to additions to the Java virtual machine (JVM™), inevitably affects the extent to which a mechanism can support the full language and the performance that can be achieved. In addition, although the mechanism itself can leverage the portability of the Java platform, application development and deployment often become correspondingly more complex.

This paper has two purposes: first to define a set of criteria by which persistence mechanisms can be judged and compared; second, to use the criteria to compare a representative set of the mechanisms that have been developed to date.[1] Note that it is not our intention in this study to compare competing implementations of the same mechanism from different providers.

2. Persistence in the Java Platform

The first version of the Java Language Specification (JLS) [GJS96] did not address object persistence directly; the only reference to the concept appears in the specification of the `transient` modifier, a keyword that was essentially reserved for future use. With each evolution of the Java

1 The author was involved in the development of one of the reviewed mechanisms, orthogonal persistence.

platform, the range of available persistence mechanisms increased, and we will briefly review the history of these developments.

The initial release of the Java platform, JDK™ 1.0, provided very limited support for persistence. There was the conventional mechanism to encode and decode basic types using input/output streams, that could be connected to files in an external file system. There was also support for encoding and decoding a table of properties (`java.util.Properties`) to and from a stream.

With the JDK 1.1 release came support for Java Object Serialization (JOS) [Sun99], a mechanism that supports the automatic encoding and decoding of nearly any object to and from a stream, respectively. Unlike the property table encoding, which is textual, object serialization uses a standard binary encoding format. JOS is effectively the default persistence mechanism for the Java platform, and the second edition of the JLS [GJSB00] refers to it in the specification of the `transient` modifier. JOS is also used as the argument marshalling protocol for Java™ Remote Method Invocation (Java RMI) [Sun96]. At its lowest levels, JOS builds on the platform support for encoding and decoding basic types. As we shall discuss later, serialization suffers some serious problems with class evolution, and this led to a new system called "Long-term Persistence for JavaBeans™" that debuted in Java 1.4.

JDK 1.1 also introduced the Java™ Database Connectivity (JDBC) API [FEB03] as a standard way to communicate with a relational database through the SQL language. JDBC provided the springboard for a major assault by the Java platform on the enterprise computing domain and arguably is by far the most successful API in the Java platform. The designers were remarkably successful in marrying the Java programming language to SQL and, consequently, JDBC is very easy to use for straightforward database access. However, in the situation where the application requires an object-oriented view of the underlying data, for example, transforming foreign key relationships into equivalent inter-object references, the programming is considerably more complicated and error-prone. To address this issue many systems have been developed that attempt to automate the process, which is generally referred to as object-relational mapping.[2] Most development environments provide some degree of support for this process. The JDBC API continues to evolve and has recently added support for directly mapping a class in the Java programming language to the equivalent type in the object-relational extensions to SQL.

In parallel with the JDBC development, the Object Database Management Group (ODMG) defined a binding of its object model to the Java programming language [Ced96], and several object-database vendors provided implementations of the binding. Since object-databases are much less widely deployed than relational databases, and are less standard, which is reflected in the ODMG binding, this exercise was less successful. With the advent of the Java Community Process℠ (JCP), the focus of this effort was replaced by a broader proposal, Java™ Data Objects (JDO) [JR03]. JDO aims to support a wide range of underlying persistent stores, including relational databases, while presenting an object model to the programmer that is similar to that defined by the JLS, but with a small number of restrictions and a few additions. Many of the object-database vendors now support JDO in their products.

Paralleling these efforts was the collaborative research project between Sun Microsystems Laboratories and Glasgow University to specify and prototype orthogonal persistence for the Java platform (OPJ) [JA00]. This approach aimed for complete support for all aspects of the language with a very high degree of transparency. In essence OPJ added support for stable memory to the JVM, and the implications of this requirement on JVM implementations ultimately prevented its acceptance.

2 Not to be confused with Object-Relational databases.

Finally, the success of the Java platform in the enterprise domain, begun by JDBC, led to the development of Enterprise JavaBeans™ (EJB), an ambitious attempt to marry a component model for application development with support for distributed transactions and legacy datastores. EJB forms part of the Java Enterprise Edition (J2EE™) [Sun03a] and is supported by all the major vendors. EJB provides transparency for persistence and transactions at the cost of a rigid and complex framework that limits the use of many of the standard features of the Java programming language.

3. Comparison Methodology

This section describes the criteria and methodology that we used for comparing the persistence mechanisms.

3.1 Criteria

We take an object-centric approach to the criteria that we use to compare the persistence mechanisms. We assume that the application is using the Java platform because it provides an appropriate set of facilities. In particular, we expect that the application is well matched to the Java object model, and that this is exploited in the essence of the application. There are applications for which this is not the case. For example, it is possible to use the Java graphical user interface components in an application that deals directly with a relational database. In this case the essence of the application is not object-oriented and our criteria would be inappropriate.

The criteria we chose are the following:

- **Orthogonality**: Do all values have equal rights to persistence, regardless of their type?
- **Persistence Independence**: Does the code have to be modified to work with the persistence mechanism? Does the application build process have to be modified?
- **Reusability**: Is code written under one persistence mechanism easily reusable under another?
- **Performance**: How is performance of the application affected by the persistence mechanism?
- **Scalability**: How does the mechanism perform as the size and number of objects increases?
- **Transaction Support:** How does the system support the classic requirements of atomicity, consistency, isolation and durability (ACID) of the persistent objects? Does the system permit variations on the ACID model, for example, nested transactions?
- **Operational Complexity**: Does the persistence mechanism require additional work to support a deployed application?
- **Evolution:** How well does the mechanism support changes to the type and behavior of the objects?

Note that whether a persistence mechanism provides a query language is not one of the criteria. We take the view that a (special) query language is not directly relevant to a persistence mechanism. Rather, it is simply another tool for computing with (persistent) objects. From our object-centric perspective, the Java programming language is the query language, just as SQL is the query language for the relational perspective.

3.2 Methodology

Although one can form some conclusions on how a system supports the above criteria from the documentation, a meaningful comparison requires actually testing the systems on a range of example applications. This is due to the fact that, in this domain, the devil is so often in the details that may not be immediately apparent from the specification. Standard benchmarks can form part of this set, but real applications are much more valuable. Unfortunately, it is not easy to acquire the latter, and they tend to be written to target a specific persistence mechanism.

For this study we chose the OO7 benchmark [CDN93], [CDKN94], which was originally written to compare object-oriented databases, and attempts to simulate typical CAD applications. OO7 has been criticized as a useful benchmark [TNL95], on the grounds that, in practice, it cannot be configured to represent real CAD applications and does not provide adequate insight into the performance of an OODB. While this may be true, its data structures are sufficiently representative of typical object-oriented applications and it is widely known and reported. Since we are using it to compare quite different persistence models and implementations, and not for comparative OODB studies, it is adequate for our purpose.

3.3 Systems Compared

We chose to compare the systems that were briefly described in section 2, because they effectively cover the spectrum of widely-available persistence mechanisms:

- Java Object Serialization (JOS)
- JavaBeans Persistence (JBP)
- Orthogonal Persistence (OPJ)
- Java Database Connectivity (JDBC)
- Java Data Objects (JDO)
- Enterprise JavaBeans (EJB)

To avoid licensing issues in reporting results from commercial systems, we use identified reference implementations of the above mechanisms wherever possible. JOS and JBP are included in the J2SE platform. For OPJ we use the PJama [AJ00],[LMG00] system. The same database server and JDBC driver is used for all persistence mechanisms that require a SQL database. Two variants of JDO are studied, the reference implementation [Sun03b] that uses a file-based custom object store and the Transparent Persistence module of Forte™ for Java™, release 3.0, Community Edition [Sun01], that maps to SQL. The EJB variants were tested on a variety of J2EE servers, and performance results, which varied widely, are reported for a leading production quality server.

4. Discussion of the Comparison Criteria

In this section we discuss the comparison criteria in more detail.

4.1 Orthogonality

In programming language terminology, the principle of orthogonality is sometimes referred to by the term *first-class*. It means that, for any one language feature, any of the other (orthogonal) features of the language can be combined with it uniformly. Orthogonality is generally considered

desirable in a language, as it produces a simpler and cleaner design. However, most languages fail to support it completely, typically for reasons of implementation difficulty or performance concerns. The consequence is special cases that surprise new programmers that may lead to contorted application designs.

Since the JLS takes no strong position on persistence, we are breaking new ground if we add persistence as an orthogonal language feature. However, the arguments of the last paragraph justify this position if the full power of the language is to be available.

Java already exhibits some lack of orthogonality in its type system that will become clear in the following overview. Java provides familiar basic types, for example, `int` and `float`, that provide a fixed set of operations and also supports user-defined object types that inherit from `java.lang.Object`.[3] Many operations that are possible on object types are not available for the basic types and vice versa. This is partly mitigated by the provision of wrapper types, such as `Integer`, but these are cumbersome to use.[4] Java also provides arrays, which share some characteristics of the basic types and object types. For example, an array is assignable to `Object`. Java also provides exceptions which, while having a very restricted set of operations, share characteristics with object types. Finally, and significant for the orthogonality of persistence, Java provides the `Thread` type for concurrency and the `Class` type for run-time type reflection.

Certain aspects of all the Java types require special support by the JVM. It is in the provision of this special support that the orthogonality of persistence and type becomes problematic.

For example, threads require special and typically quite complex support from the JVM. To make a thread persistent requires that all the state associated with the thread, much of which is not explicitly visible to the Java programmer, can be captured and saved. Of the mechanisms compared, only OPJ aspires to support persistent threads.

Similarly, every object type has an associated `Class` object, that defines its behavior and extends to include the code of the methods. This association is an invariant of the object that cannot be changed by the programmer. Orthogonality of persistence and type requires that this invariant be maintained for the lifetime of the object which, again, requires considerable virtual machine state to be captured and saved. Maintaining this binding also causes non-trivial evolution problems because the code and data cannot easily be separated.

Java's dynamic class loading facilities cause particular problems for layered persistence mechanisms. Any persistence mechanism that requires the set of classes whose instances may be made persistent to be identified before the application runs, will be unable to deal with classes whose existence only becomes known at run-time. Similarly, the Java type system distinguishes classes with the same name, but loaded with different, user-defined, classloaders. Again, significant virtual machine state has to be made persistent to capture this binding.

The principle of orthogonality applied to `static` variables, would permit the variable to be either transient or persistent. In an early version of the JLS, the combination of the `static` and `transient` modifiers was explicitly prohibited, thus violating orthogonality. This was subsequently rescinded but, as seen later, many systems treat `static` variables as implicitly transient. The reason is strongly correlated with the persistence of class instances, since static initializers are executed as part of the class loading process.

In summary, without support from the JVM, implementing full orthogonality is impossible.

---

3 From now on, we will supress the `java.lang` package prefix.

4 This is remedied by the automatic "boxing/unboxing" facilities of J2SE 1.5.

4.2 Persistence Independence

Orthogonality addresses the basic question of whether a particular type of object *can* be made persistent. Persistence independence addresses what a developer actually has to *do* to achieve this for a given type.

There are two aspects to this. First, is it necessary to make any source code changes? This includes annotations to the code in the way of special comments. Second, is it necessary to use a special development process, e.g., a special compiler, pre-processor or post-processor?

Some mechanisms claim persistence independence by answering “no” to the first question, but “yes” to the second. While it is true that modifications to the development process are easier to handle than edits to the source code, they can significantly complicate matters. One often neglected area is the impact of post-processing techniques on source level-debugging, because the debugger operates with the transformed program, which the programmer may not be able to see.

4.3 Reusability

This is closely related to persistence independence, since any mechanism-specific modifications to code, whether directly or indirectly, may likely render that code unusable in another context. At the very least, any external dependencies introduced into the code will have to be satisfied in the new context. This has the potential to limit the Java platform’s “write-once, run-anywhere” proposition because, for example, post-processed bytecodes may require certain non-standard packages to be available.

4.4 Performance

Every persistence mechanism has an effect on performance. There are two measures of interest. First, the total cost over a complete program execution and, second, how this cost is distributed over time. Generally, the more orthogonal a persistence mechanism, the more the cost will be spread out over the whole execution. This is similar to the impact of automated main-memory (heap) management, which typically distributes the allocation and collection costs but, in doing so, may slow the application, for example, by inserting write barriers.

4.5 Scalability

The Java platform, albeit with some standardized subsetting, scales from smartcards to supercomputers. Ideally the persistence mechanisms should scale similarly. Scaling in the large is measured primarily by the total number and size of persistent objects that can be handled. In the small, it relates more to the minimum footprint, in code and data, that the system requires to operate. Historically, scaling in the large was seen as the critical metric for persistent systems. However, due to the Java platform's growth in the embedded systems area, minimum footprint is also important. Generally the impact of a persistence mechanism on the data space (heap) is more significant than its impact on the code size. This is because the additional code size is proportional to the number of loaded classes, whereas per-object space overheads scale are proportional to the number of object instances. It is not unknown for a persistence mechanism to add a 20% overhead to the typical object size of about 40 bytes.

4.6 Transactional Support

Since persistence is fundamentally concerned with the preservation of state over long periods, and because most computer main memories are volatile, a persistence mechanism must address

the *durability* of the objects made persistent, and the *atomicity* (all or nothing) of the mechanisms that alter this preserved state in the face of hardware or software failure. In database systems, these characteristics are addressed by the notion of a *transaction*. Two additional characteristics, *isolation* and *consistency*, are also associated with the transaction concept, yielding the so-called ACID properties. Isolation and consistency are not fundamental to a persistence mechanism; indeed, the Java programming language already has a separate mechanism for isolation, namely synchronized methods and blocks.

Generally, transaction mechanisms hide the use of locks from the programmer, whereas thread mechanisms expose explicit locks in the source code. There is therefore potential for conflict between the two mechanisms. The consistency property is typically defined rather loosely. In a relational database, consistency is partly equivalent to the invariants that are routinely enforced by the Java programming language type system. Semantic consistency is partly enforced by declarative constraints and partly by application logic. In a Java application, semantic constraints are explicitly programmed in class methods. This is a strong argument for maintaining the binding between an object instance and its class in the persistent store.

Most of the systems under comparison support a transaction model, usually simple ACID, and provide an associated API. The interaction between transactions and threads is often unclear and prone to semantic ambiguities. Many of the systems also allow access to persistent objects from multiple JVMs, which further complicates matters, since multiple copies of an object may exist simultaneously in several JVMs, or, in some cases, in the same JVM. The multiple copies trivially provide isolation, and also permit varying implementations of transactional locking. For example, one system might implement pessimistic locking (database transactions), causing affected applications to block when trying to access an object, whereas another system might employ optimistic locking and only detect conflicts at transaction commit time.

4.7 Operational Complexity

This criteria captures the amount of work that arises directly or indirectly from the persistence mechanism when an application is deployed and executed. It includes the management of resources such as file systems, raw disks, operating system processes, and any additional complexity added to the environment of the Java platform, such as special classpath settings or run-time properties.

4.8 Support for Change

Since change is the norm in computing, it is important that a persistence mechanism support orderly change of the objects under its control. Support for change divides into two models, *evolution* or *versioning*. In the evolution model, which is by far the most common, a change is (logically) applied to all objects simultaneously and is irreversible. Sometimes the change to an individual object is applied lazily on demand. However, the old state is never visible in normal use since any attempt to access the object either causes it to evolve or causes an error. In contrast, the versioning model supports access to old states of an object if required. Effectively, it is as if an object is always viewed through a filter that selects a particular version. The versioning model has many possible realizations. In particular, it may or may not be possible to access more than one version of an object simultaneously.

While many changes may be handled automatically, in general, in either model, certain kinds of changes will require that special transformation code can execute in both the old world and the new world, if only briefly, in order to effect the change.

Support for change places a significant demand on the implementation of a persistence mechanism, which explains why it is frequently absent or limited in scope. In contrast this is one of the most notable strengths of a relational database – e.g., the ability to add columns to a table in a live database. However, the relational database provides no assistance towards achieving orderly change for the whole application set, specifically the programs that access the data.

5. An Overview of the Systems Compared

5.1 Java Object Serialization (JOS)

This mechanism is widely known due to it being part of the standard platform, so we will not describe it in detail. JOS, like many persistence mechanisms, is reachability-based, starting from a single root object. The serialization includes all objects reachable, transitively, from the root, by following those fields of an object that are references to other objects. A representation of each object, including its class, and its fields, is written to the output stream. Deserializing is achieved by reading a previously written stream. The information on the class is limited to the name and an encoding of the method signatures; in particular, the bytecodes of the methods are not saved.

Sharing is preserved within a single[5] serialization, that is, only one copy of an object will be written and any sharing will be re-established on deserialization. Serialization is similar to message passing in that copying is an intrinsic aspect of the procedure. In particular, object identity and hashcode are not preserved when an object is serialized and then subsequently deserialized, nor is sharing preserved across multiple serializations.

JOS provides a general mechanism for the implementer of a class to interpose on the serialization process, which can be used, amongst other things, to transform or omit certain fields of the object. In practice this must be utilized much more than one might expect to overcome scaling issues inherent in the JOS specification.

Only classes that implement (directly or indirectly) the (empty, i.e., tagging) `java.io.Serializable` interface can be serialized. This is intended to prevent unintended serialization of sensitive data.

5.2 JavaBeans Persistence (JBP)

In contrast to JOS which takes an implementation-oriented approach to persistence, JBP takes an interface-oriented approach. The initial drive behind JBP was the extreme difficulty experienced by the Swing user-interface community in using JOS to persist components that could be read successfully by the ever-evolving versions of the Java platform. Essentially, the limited support for class versioning in JOS was unable to cater to the extent of the evolution of the Swing components.

JBP takes advantage of the fact that a JavaBean is required to expose its public state via “getter” and “setter” methods. If all state that contributes to the observable behavior of the object is exposed in this way, then it suffices to record the values of these properties, as returned by the “getter” methods, in the persistent state, confident that an equivalent object can be created later by calling the associated “setter” methods. The virtue of this approach is that the mechanism is based on the class interface and decoupled from the implementation of the class, allowing its un-

5 The scope of the sharing has a subtlety in that it applies to the underlying stream rather than the call to write the object graph. The sharing applies to a sequence of calls until a flush or close operation is applied to the stream.

fettered evolution. The interface generally evolves more slowly and usually in a backward compatible way.

JBP also includes an ingenious scheme to minimize the size of the persistent representation that avoids recording properties that have their default values, although this comes at the price of cloning the initial data structure, which causes a serious scaling problem with large amounts of data.

In principle, the format of the persistent representation is arbitrary, and several variants were developed initially. However, the current release only includes an XML-based file format.

Since many classes do not follow the JavaBeans conventions completely, a mechanism is provided to interpose and customize the process. Unlike JOS which requires the customization methods to be added to the class with special method names, JBP takes a less intrusive approach and consults a table of registered class customizers dynamically at run-time.[6]

Although initially targeted at the Swing subsystem, as of the 1.4.1 release, JBP is now supported for all core classes in the platform.

5.3 Orthogonal Persistence (OPJ)

OPJ is specified as an extension to the JLS that provides orthogonal persistence for the Java platform, following the principles defined in [AM95]. These principles are, not surprisingly, closely aligned with the criteria for this study. A series of prototype implementations of OPJ, called PJama [AJ00], [LMG00], were developed as extensions to production JVMs and made available for research and evaluation purposes.

A PJama system consists of a variant of a standard JVM (PJVM), modified to support orthogonal persistence, and a persistent store, an entity that exists in stable storage and contains the persistent state of the computation that is executing on the Java virtual machine. The PJVM is itself stateless; all information that is needed to resume execution is maintained in the stable store. The state of the system is checkpointed periodically to the store either by an explicit call from an application, or when the PJVM exits normally. The checkpointing is atomic and an exception or crash leaves the store in the state at the last successful checkpoint. In the studied implementation, the stable store is implemented as a disk file, although this is not visible to the application. Indeed, a consequence of the OPJ model is that it is the PJVM that opens the store and not the Java application.

The illusion that OPJ creates is that of *continuous computation* for the entire PJVM, although in practice the persistent state of the computation follows a sequence of discrete states with an interval determined by the checkpoint frequency. The scope of the persistent computation is limited to state managed by the PJVM and application intervention is required to capture state that refers to objects outside the scope of the PJVM, for example socket connections. A simple API is provided for this purpose.

Given the incomplete support that is characteristic of the other persistence mechanisms, it must be stressed that OPJ makes persistent, conceptually at least, *everything* that is needed to resume the computation, as if it had never been suspended. Some important corollaries of this fact are:

- A class is loaded and initialized exactly once regardless of the number of times the computation is suspended and resumed.
- The binding between an instance and its associated `Class` object never changes.[7]

6 At a performance cost over the JOS approach.

7 Modulo explicit mechanisms provided for class evolution.

- Execution state, represented by instances of class `Thread`, is made persistent.
- Automatic (object) memory management is extended to the persistent domain.

The OPJ model demands some additional tools and mechanisms that are not required in a standard Java platform, notably for the management of class evolution and persistent store garbage collection. The PJama implementation provides simple tools to support these activities.

5.4 Java Database Connectivity

JDBC is an API that provides access to a database server that supports the SQL language. JDBC makes good use of the features of the Java platform, such as automatic memory management, convenient string-handling, and the collection classes to simplify programmatic access to SQL. Transparent mapping of basic data types such as `String` or `int` into the corresponding SQL types is also provided. JDBC actually defines a set of "JDBC types" that act as portable intermediaries between the Java types and the SQL types, and are mapped transparently to the underlying database type by te JDBC driver.

JDBC manages the complexity of maintaining database connections, and also provides a range of mechanisms to help performance and scalability. For example, "prepared" statements can be reused many times with different arguments and are typically compiled and cached by the SQL server. Statements may also be batched together to reduce the number of round-trips to the server.

To help deal with the variations of different SQL databases, a rich metadata interface is provided that allows applications to discover at run-time exactly what features a particular database supports, and to access the database schema.

The JDBC API, currently at version 3.0, continues to evolve. Recent improvements include statement pooling, whereby prepared statements may be cached for reuse by multiple logical connections, and better support for advanced features from the SQL 99 standard.

5.5 Java Data Objects (JDO)

JDO is an attempt to provide "transparent persistence" for developers of Java applications, across a wide range of underlying datastores including relational databases, object databases and other custom datastores. It is an outgrowth of the work that was begun by the ODMG on the binding between the Java platform and object databases.

JDO therefore shares some similarities with OPJ in that both aim for transparency. The essential difference is that JDO explicitly introduces the existence of an external datastore, that may be accessed concurrently by unknown third-parties, and that an explicit mapping exists between the JVM environment and the external store. OPJ, in contrast, addresses the stability of the Java object memory and associated JVM state. This difference is reflected in two key aspects of the JDO specification, namely the introduction of an additional form of object identity, JDO-identity, that subsumes JVM object-identity, and the specification of an object-model life cycle that introduces states, potentially visible to an application, such as "hollow," where the object fields are not filled in from the corresponding data in the datastore. JDO-identity comes in three forms: *Datastore*, which is managed by the external store and unrelated to the fields of an object; *Application*, which is managed by the application and based on certain (primary-key) fields of the class; and *Nondurable*, which is managed by the JDO implementation for data that has no natural unique identifier, for example entries in a log file.

JDO is not transparent for all data types and explicitly introduces first-class and second-class objects for persistence purposes. These transparency limitations arise from two sources: firstly from the constraints imposed by the underlying datastores, which have to be treated as immutable legacy systems, and secondly from the fact that a JDO implementation must treat the JVM as a black box. In particular, although arrays in the Java programming language have some object-like features, they cannot be handled by the standard JDO bytecode post-processing ("enhancement") as they are not full-fledged classes in the JVM. In consequence, arrays are second-class objects in JDO, and must be encapsulated inside a (persistent) class. In particular, if such an array is passed outside the class, modifications to the array will not be detected by the JDO implementation, leading to incorrect behavior.

JDO includes a query subsystem that is accessed programmatically via the Query class. It operates over collections of objects and allows simple boolean conditions to be expressed using Java programming language syntax. JDO provides a way to access the entire extent of a class, and this typically forms the starting point for queries. JDO also allows objects to be looked up using JDO object identity, which is the other way to locate objects.

JDO defines a standard way to describe the details of the persistence of a class or package specified in XML. At a minimum this must specify which classes can be made persistent but may include details such as exactly which fields are to be persisted.

JDO defines the PersistenceManager class as the main client interface for controlling the system. Instances of PersistenceManager are themselves created by a PersistenceManagerFactory class that is created either from a properties file or by explicit instantiation in the code. Object creation and access must run inside a transaction bracketed by the begin and commit methods of the Transaction class. The current transaction can be accessed from the PersistenceManager instance.

5.6 Enterprise JavaBeans

EJB is a component architecture for the development of distributed business applications and forms part of the Java™ 2 Platform Enterprise Edition (J2EE™ platform). In exchange for following the EJB architecture, developers are provided transparent support for distribution, persistence, transactions and security. There is also the promise of a "write-once, run-anywhere" guarantee for EJBs, similar to that for standard Java classes, in any J2EE-compliant implementation. Beans are deployed in a *container* that forms part of a J2EE implementation, provides life-cycle support for the bean and interposes on all access to beans.

Beans come in three forms: session, entity and message. Message beans are not relevant to this study.

Session beans model workflow and typically support a specific client that is interacting with the system. Session beans are not intended to model persistent data although they may access datastores and may survive crashes. Session beans come in two forms, stateless and stateful. The former carry no data specific to a particular client and can be pooled and used for any invocation, whereas the former carry client-specific state and are bound to that client for the duration of the session. Since session beans do not model persistent data, we do not consider them further.

Entity beans are intended to correspond to persistent data, typically a row in a relational database table[8] and have strong availability guarantees in the face of system failures. The persistence of entity beans can be managed directly by the bean (bean-managed) typically using JDBC, or managed automatically by the container (container-managed). Container-managed persistence might itself be implemented using a mechanism like JDO.

8 This fine-grain mapping is now deprecated for performance reasons.

The EJB framework is relatively complex and involves the creation - at a minimum - of two interfaces, called *home* and *remote* and also an implementation class, referred to as the *bean* class. This set is collectively referred to as the bean. The home interface is used for locating or creating bean instances and the remote interface is used by clients to invoke a bean's business methods. The bean class provides implementations for (some of) these methods as well as a number of callback methods required by the framework. The J2EE implementation will typically provide additional implementation classes, for example, of the remote interface.

Similar to JDO, EJBs follow a defined life-cycle, but it is more visible to the programmer because it is reflected by required callbacks from the container to the bean implementation class. The programmer-provided bean implementation class must implement these callbacks. The life-cycle includes an explicit *pooled* state, which corresponds to an allocated but unassigned bean class instance. Pooled instances are activated as needed, for example, when a new persistent entity is created or when retrieved by a *finder* method from the underlying datastore. To support scalability, the life-cycle also includes *passivation* and *activation* transitions, that allows a bean instance to be freed up, similar to memory paging in a virtual memory system.

EJB provides a query language, EJBQL, which can be used to implement finder methods directly. EJBQL supports the relational model, with some syntactic sugar to close the gap with the Java programming language.

In addition to the bean code, a developer must also provide a deployment descriptor, in XML, that provides information on the bean. This is used by bean assemblers and bean deployers and can be used to customize the bean to a particular environment. In EJB 1.0 this information was coded in the Java programming language and used JOS as the persistence mechanism. The switch to XML in EJB 1.1 is indicative of the general trend away from JOS to XML.

At the time of writing, proposals for EJB 3.0 were just being released. These call for a dramatic overhaul of the entity beans specification that will move closer to JDO in spirit, if not in detail.

6. Applying the Criteria

In this section, we discuss the comparison criteria applied to each system in turn.

6.1 Java Object Serialization Analysis

JOS: Orthogonality

JOS scores lower on orthogonality than might be expected, in part because it was added to the platform after the initial release. There was, therefore, the question of the status of code that had already been written and deployed. In the initial early access version of JOS, an object of any class could be serialized, unless the implementor of the class explicitly prohibited it. Since persistence by reachability can lead very easily to unexpected objects being included, this was deemed to be incompatible with Java's privacy and security guarantees. For example, it might lead to some crucial data, stored in a private field of a class, being written to the file system or transmitted over a network. Since prior Java code might be affected by this, the requirement to implement the java.io.Serializable interface was added in the public release. This is, by definition, a violation of orthogonality, since it partitions the set of classes into two and requires foresight by a developer. Ironically, a class can inherit serializability, so a developer must still be on guard to protect sensitive information. Absent such considerations, the only sensible choice for a developer is to declare serializability, which leads to its routine appearance in code. Also, many systems, for example, EJB, mandate that certain classes are serializable.

However, many classes in the standard platform are not serializable, notably Thread, and many AWT classes. A serialized instance does carry certain information about its class, enough for compatibility checks during deserialization, but because the bytecodes are not saved, precise recovery of the the behavior depends on classfiles remaining consistent between runs. Since static variables are effectively associated with a Class instance, these are not serialized automatically with instances of the class. However, static variables that inherit from Object can be serialized manually by an application in an extra phase if necessary.

JOS: Persistence Independence

The requirement for a class to statically implement the java.io.Serializable interface violates persistence independence. JOS has no impact on the application build process.

JOS: Reusability

The use of JOS generally does not prevent reuse in other mechanisms. The exception to this is the unfortunate decision to assign serialization-specific semantics to the transient keyword that occurred in JDK 1.1. The effect is that, in some cases, transient means "user-defined serialization," rather than "not persistent" or "not written to the stream," as suggested by the JLS. The JOS interpretation renders the keyword unusable by other persistence mechanisms, because fields marked transient for the serialization idiom would also be interpreted as transient by the other mechanisms, with disastrous consequences. In practice this idiom is widely used to transform data representations during serialization, notably in the standard collection classes. For example, the idiom is used in such a way that the fields denoting the contents of the standard hash-table class are declared transient! An alternate mechanism that explicity denotes the fields to be serialized by a static array field in the class named serialPersistentFields was eventually provided, but this does not seem to have been adopted widely, probably due to the extra programming effort involved.

JOS: Performance

Performance is adequate for relatively small object collections, with only a modest overhead compared to any hand-crafted approach. There is no cost to the application during normal processing, only during the serialization step itself.

JOS: Scalability

Scalability is a problem for serialization. A direct consequence of its copy model is that it suffers the "big-inhale/exhale" problem as the object graph increases in size. This can only be solved by the programmer explicitly breaking up the graph into smaller pieces, at which point much of the simplicity disappears and the old problems of maintaining object sharing and global consistency resurface for the programmer to handle.

Another scalability problem relates to the depth-first processing of object graphs. Deep structures, for example, long lists, can cause stack overflows when serialized. This is the main rationale for the custom serialization of the standard collection classes.

JOS: Transactional Support

The durability of the serialized objects depends on the sink that the stream is attached to. If, as is typical, the sink is a file, the durability is that associated with the underlying file system. Atomicity is not guaranteed, unless the file system is itself transactional, which is not typical.

There is no support for isolation from other threads that might be mutating the objects while the serialization step is taking place. This renders the mechanism suspect in a system of non-cooperating threads. However, the nature of serialization, which requires that a programmer reason carefully about reachability to avoid including unwanted objects, tends to define an intrinsically isolated group of objects.

JOS: Operational Complexity

The only operational complexity associated with serialization is that of managing the external files that are needed to store the serialized state.

JOS: Support for Change

The first release of JOS had no support for change, but a simple versioning system was added in the first revision. This was based on a similar philosophy to the binary compatibility rules already defined by the JLS, and assumes that classes evolve in a mostly additive manner. For example, deleting a field between class versions is not permitted. In order to avoid two unrelated classes with the same name accidentally being considered versions by the system, a version of a class must define a constant SERIALVERSIONUID with a value obtained from a hash of the original class.

In practice, the compatibility rules proved too limiting for the kinds of changes being made between, for example, releases of the Java platform, essentially due to the fact that there is pressure for implementation classes to evolve in ways not permitted by the overly simple compatibility rules.

6.2 JBP Analysis

JBP: Orthogonality

If a developer follows the JavaBeans conventions, and arguably there are good reasons for doing so independent of the JavaBeans framework, then orthogonality is very closely approximated. The complete support for core classes is an improvement over JOS. However, code that does not follow the conventions may need to be modified unless the provided set of persistence delegate mechanisms can be used to work around problems.

The fundamental requirement imposed by JBP is that there be some way to recreate the state of the object solely by using the methods provided in the public interface to the object.

JBP: Persistence Independence

JBP is an improvement over JOS in this area. There is no requirement to indicate persistence by implementing a special interface, and the customization code that is needed for construction can be separated from the class.

JBP: Reusability

JBP does not impact the reusability of code.

JBP: Performance

JBP has no impact on the run-time performance of code. The reference implementation uses XML to encode the persistent data. Processing XML is relatively expensive, so reading and writing the persistent data is relatively slow. The implementation also utilizes the Java platform's reflection system that also adds overhead.

JBP: Scalability

The reference implementation of JBP constructs a copy of the object graph during output to support the optimization of not writing out fields with default values. This obviously limits scalability.

JBP: Transactional Support

The remarks for JOS apply equally to JBP.

JBP: Operational Complexity

The remarks for JOS apply equally to JBP.

JBP: Support for Change

There is no explicit support for change in JBP. However, the fact that the persistent representation is stored in a public XML format, makes possible arbitrary transformations using a wide variety of tools.

6.3 OPJ Analysis

OPJ: Orthogonality

OPJ was designed to meet the criteria that we are using for comparison. All types can be made persistent, without any modifications to either source code or class files.

OPJ: Persistence Independence

No source code modifications are required except to explicitly define a root of persistence, which is usually localized to per-application initialization code, or to explicitly force a checkpoint (transaction commit).

No modifications to the development process are required beyond using an OPJ virtual machine for testing. No source pre-processing or classfile post-processing steps are need and any compiler can be used. Since almost all code can be developed without reference to the small persistence API, it is possible to use an off-the-shelf development environment except for the final testing phase.

OPJ: Reusability

Code developed for a standard Java application can easily be reused in an OPJ application and vice versa.

OPJ: Performance

An OPJ virtual machine typically adds a consistent overhead to any application because of the checks for non-resident objects that occur on all object member access. In essence, this is the price for the high degree of orthogonality. The measured overhead is dependent on both the JVM implementation and the application and has been shown to average 15-20% in the PJama prototypes. From the perspective of the JVM implementor and SPEC benchmarks, this overhead seems very significant. However, it pales in comparison to the overheads of other persistence mechanisms as shown later.

The overhead can be measured quite consistently when all of an application's objects fit in the object cache, but becomes much more variable when object swapping occurs, owing to disk latencies. Interestingly, on occasion, applications whose data fits completely in main memory have been observed to run faster than under a standard JVM, presumably due to hardware caching effects related to the increased locality of the objects.

OPJ: Scalability

There are no inherent scalability issues for an OPJ JVM, since all the persistence mechanisms that might affect scalability are implemented within the JVM itself. In particular, memory management aspects, such as persistent object caching, can exploit the low level facilities of the JVM.

OPJ: Transactional Support

Using standard transactional terminology, an OPJ virtual machine supports a "chain" transaction model, with no programmable rollback. A transaction is implicitly started when the virtual machine starts or resumes execution – the transaction is commited by the checkpoint operation after which a new implicit transaction is immediately begun. Although an implementation might maintain a history of checkpoint instances, the specification does not require this. The only way to abort (single-level rollback) is to exit the virtual machine with an error.

OPJ: Operational Complexity

Operational complexity is low which can be attributed largely to the simple architectural structure of an OPJ system, in particular, the simple binding of one virtual machine process and a persistent store. For most applications, operational complexity is actually lower than for an ordinary Java application because there is no need to manage the application classes using a file-system classpath. Once the classes have been made persistent in a store, which can be handled once in an initialization step, the application can be completely self-contained in the store file. In particular, it is possible to distribute an application as a pre-initialized store. Managing the store file is the only significant operational overhead.

OPJ: Support for Change

The OPJ specification makes no explicit provision for the support for change. However, the fact that the specification mandates that the binding between code and data be preserved in the persistent store, enables tools that can reason about the existing and proposed states and also execute code associated with both states. The PJama system provided a special API to achieve evolution, that might, after more experience, have become part of the standard specification [DM01].

6.4 JDBC Analysis

JDBC: Orthogonality

JDBC makes no pretense of orthogonality - it is intended explicitly as a bridge between the Java programming language and the SQL language. Conceptually, data conversion always occurs as data crosses the bridge in either direction, although conversion is transparent for simple data types.

This complete separation and explicit conversion, although somewhat tedious to program, has the great virtue of clear semantics, with the programmer in complete control of when data is transferred between the domains.

JDBC: Persistence Independence

Again, there is no pretense - code is completely specific to JDBC.

JDBC: Reusability

JDBC code is only reusable in another JDBC context, and, then, provided that the database vendor supports the same dialect of SQL. Beyond simple applications this proves to be difficult, typically because the actual SQL code is defined in the context of a particular database vendor without adequate regard to application portability.[9]

JDBC: Performance

The performance of JDBC is dominated by crossing the domain boundary between the Java application environment and the SQL environment. Object-style navigation of data structures represented in SQL is extremely slow as a query usually involving a SQL SELECT is needed for each navigation step, which can be several orders of magnitude slower than the equivalent pointer-following operation of a JVM.

The key to good performance is to execute the majority of the application logic as SQL queries, thereby leveraging the query optimization mechanism of the SQL server and minimizing the number of round trips between the JVM and the SQL server.

JDBC: Scalability

The principal scalability issues for JDBC programming are connection management, the number of round trips to the server and the amount of information that flows across the domain boundary.

Connection management is important because connections are expensive to set up and consume resources such as socket connections and database processes. This issue also affects reusability as connection management might vary with different deployment environments and should be factored out of reusable code. JDBC supports connection management through pooling of connections whereby the JDBC driver maintains a number of open connections, sharing them transparently between clients in the same JVM. Conceptually, the client always closes the connection after each interaction with the database server, but a pooled connection will, in fact, be held open between calls and shared between clients.

9 Changing the database vendor is as infrequent as changing the hardware or operating system vendor.

JDBC provides batching mechanisms to minimize the number of round-trips to the server, allowing several separate SQL queries to execute in one round trip. This isn't transparent to the programmer, however, and its usefulness may depend on the application semantics.

JDBC also provides several mechanisms for controlling data flow across the domain boundary, although this involves a trade-off on round trips. Large objects (BLOBs) can be streamed across as required and large rowsets can be controlled through iterators.

In general, JDBC applications that do not create large numbers of objects that proxy data from the database, scale very well as they do not consume many JVM resources. In practice, however, applications typically are forced to explicitly cache data due to the high latency overhead of database round trips. This quickly becomes a source of complexity and errors, and was a contributing factor behind the development of the EJB container managed persistence framework.

JDBC: Transactional Support

By default, JDBC drivers operate in auto-commit mode, where each SQL query/update is an atomic database transaction. It is straightforward to disable auto-commit, execute several queries/updates, and then explictly commit the transaction.

Transactional considerations only become complex if the application caches SQL data in Java objects. In this case it becomes important to refresh the state of these objects at appropriate times.

JDBC: Operational Complexity

Administering a commercial relational database is a non-trivial task but for typical JDBC applications in the field the database is a given and managed by a specialist team. In the context of this study, however, the database is simply an aspect of the persistence mechanism and so its adminstration must be considered as part of the operational complexity.

JDBC: Support for Change

JDBC provides no explicit support for change. However, the fact that the JDBC API provides access to database metadata, in particular to the database schema, and that SQL provides mechanisms for changing schemas dynamically, makes it possible to write Java programs to effect the changes.

6.5 Java Data Objects (JDO) Analysis

JDO: Orthogonality

JDO supports most user-developed classes and does implement a model of persistence by reachability. However, the distinction between first-class and second-class objects violates orthogonality. In particular, arrays are not required by the specification to be first-class, which rules out most user-developed collection classes. To mitigate this, JDO specifies that an implementation must support some specific variants of the collection classes, for example, HashSet.[10] Not all platform classes are required to be supported.

The specification limits persistence to user-developed classes that implement the PersistenceCapable interface. Although it is legal to manually augment a class to implement PersistenceCapable, JDO defines a standard classfile enhancement process that achieves this automatically.

10 These classes typically are subclassed and manually enhanced to support JDO. Application-created instances are replaced transparently by the JDO run-time with instances of these subclasses.

Access to persistent objects must take place inside a transaction, which requires explicit calls to methods of the Transaction class and so has a small impact on orthogonality.

JDO: Persistence Independence

Excepting the need to demarcate transactions and establish roots of persistent objects, code can be completely persistence-independent.

However, it is necessary to follow a special development process that includes the classfile enhancement step. Failure to enhance a class that should be persistence-reachable will result in incorrect program behavior. The developer must be careful to keep the enhanced and unenhanced classfiles separate in the run-time environment. Since the enhancement process introduces additional method calls, these will be visible while debugging.

JDO: Reusability

The fact that JDORI mandates a standard enhanced bytecode format that all implementation must support means that there is enhanced bytecode portability between JDORI implementations.[11] Furthermore the format is defined in such a way that the enhanced bytecodes will operate in a non-JDO environment, albeit with a slight performance penalty.

JDO: Performance

The performance of JDO has three separate components. The first is the impact of the additional code inserted by the enhancement process. The second is the state management and persistent object cache management that is typically implemented in the Java language. The third component is the underlying external datastore, the performance of which can vary considerably. Provided that the data can fit in the heap, the performance of JDO, once the data has been read into memory and cached, should be close to the VM variant. For OO7, we measured the overhead of the JDO-enhanced bytecodes at 11.5%.

JDO: Scalability

Managing a persistent object cache in the Java programming language requires use of the weak reference facility, which can stress the JVM memory management subsystem.

JDO: Transactional Support

As noted above, JDO requires access to persistent objects to be within a transaction. Unlike OPJ, transactions are not implicitly chained, and it is between transactions that non-standard object states such as “hollow” may be observed by the application. However, JDO does support the intra-JVM transactional rollback of persistent objects.

JDO: Operational Complexity

There is a slight increase in complexity due to the need to manage enhanced and unenhanced class files. It is important that the enhanced classes are used at run-time. The XML file specifying the details of the persistent classes must also be available on the classpath.

11 As with the standard classfile format a JDORI implementation may support a customized enhanced format, but it must also accept the standard format.

JDO: Support for Change

JDO provides no support for change. Since the majority of current JDO implementations are based on mapping to relational databases or object databases, which can be evolved independently using other mechanisms, this is not a major problem. However, it would be a major issue for an implementation based on a native persistent store.

6.6 Enterprise JavaBeans (EJB) Analysis

EJB: Orthogonality

While EJB is object-centric it is far from orthogonal. EJB essentially defines a new, less powerful, object model that happens to be implemented in the Java programming language. In particular, EJB does not support inheritance. Several language features, such as threads, are disallowed in the EJB model.

EJB: Persistence Independence

Transforming an existing Java class into an EJB requires considerable changes and the end result is visibly dependent on the EJB framework. The build process is also complicated by the need to package up the EJB classes with XML deployment descriptors into jar files that are loaded by the server.

EJB: Reusability

An EJB is only reusable across EJB containers. In the context of the EJB object model, this is good. However, there is no reuse possible outside of the EJB container.

EJB: Performance

Achieving adequate run-time performance for entity beans has proven to be very challenging. So much so that the original notion that an entity bean represents a row in a database table has essentially been deprecated. Developers are now encouraged to use entity beans for coarse-grain objects, mostly to avoid performance and scalability problems [ACM01].

EJB: Scalability

The EJB architecture was designed to scale, since it was aimed at enterprise computing systems. The bean pooling mechanisms and support for passivation and activation allow a system to function when the number of entity beans exceeds the container capacity. However, much like virtual memory, if these mechanisms are actually invoked in practice, performance suffers considerably.

EJB: Operational Complexity

EJBs are hosted in a J2EE server which has a very different application model from that of a standard JVM. Applications are accessed either through other client applications or web browsers – there is no equivalent of the main method of standard Java applications.

Resources needed by the application, in particular the external database, must be configured using the J2EE server mechanisms that are, unfortunately, vendor specific.

EJB: Support for Change

EJB provides no support for change. However, since EJB is always mapped to an external, typically relational, store, the remarks for JDBC and JDO apply.

7. The OO7 Benchmark

The OO7 benchmark was developed at the University of Wisconsin to test many different aspects of object-oriented database systems. The benchmark is intended to model typical CAD/CAM/CASE applications and contains several hierarchical structures and 1-1, 1-many and many-many relationships between objects. The benchmark can be configured in a variety of ways and comes with a set of standard configurations. OO7 defines a number of different traversal, query and update operations.

The version of OO7 for the Java programming language (OO7J) was generated initially by a straightforward translation of the reference implementation in E, an extension of C++, that was provided by the designers of OO7. During this study we modified OO7J to more closely follow current programming standards, for example by using get/set methods for accessing object attributes and by using the standard collection classes of the Java platform. In consequence, it would be inappropriate to use OO7J for comparison with implementations in other languages.

7.1 OO7 Data Structures

The heart of the OO7 data structures is a “library” of *composite* parts, intended to model circuits, programming language modules or similar constructs. The size of the library is configurable. Each composite part consists of a set of *atomic* parts, intended to model indivisable entities such as a circuit element or a programming language component such as an identifier. The number of atomic parts in a composite part is also configurable. Each composite part’s set of atomic parts is private, that is, they are not shared with other composite parts in the library. Each atomic part is connected to a fixed number of other atomic parts in the same composite part. One of the connections is used to connect the parts in a ring and the remainder are randomly connected. The connections are bi-directional and indirect through a *connection* object, which carries additional information that is intended to model connection-specific data. This part of OO7 is closely based on the earlier OO1 benchmark [CS92]. OO7 adds an additional data structure, an assembly hierarchy, that is intended to model aggregate structures of components. The top-level component is a *module*; the number of modules is configurable, but is usually set to one. Each module consists of a hierarchy of assemblies, with both depth and fan-out being configurable. At the leaves of the hierarchy are *base-assemblies* that are connected to composite parts. The interior nodes of the assembly hierarchy are *complex-assemblies* that either refer to other complex-assembly nodes or to base-assembly nodes.

Both modules and composite parts reference documentation components that model large text files and are intended to test support for large objects. Each module is associated with a *manual* object and each composite part is associated with a *document* object.

Several of the objects share the same set of *design* attributes, which can be modeled by inheritance, thus providing a test for support of this language feature. One might expect any persistence mechanism for the Java platform to support inheritance, but as will be seen later, this is not always the case.

7.2 Operations

The benchmark operations fall into three groups: traversals, queries and insertions. Traversals model the traditional navigational access methods that are typical of object-oriented languages; some traversals modify (existing) objects during the traversal. Queries find sets of objects that match certain criteria and are akin to queries in a relational database. Insertions add new objects to the database. The original implementations of OO7 followed the approach of always using a query system if it was available. Initially we followed that approach, but ultimately decided to ignore the query operations altogether, based on the rationale that a query system is not fundamental to a persistence mechanism for the Java object model.

8. The Development of OO7J

OO7J is based on the version in E developed by the benchmark designers. The initial version was very faithful to the E implementation, but was subsequently modified to follow the programming conventions of the Java platform.

The goal in evolving OO7J to run on the given range of persistence mechanisms was to make minimal changes to accommodate the port to each system. Sometimes this resulted in revisiting the original version and changing a data structure to more easily accommodate a particular persistence mechanism, rather than spawning a separate variant. A good example of this is in the use of bulk data types. The original version made much use of fixed-length and variable-length arrays. Since JDO does not support arrays adequately, we modified the original version to use `java.util.HashSet`. This choice has no semantic effect on the benchmark (because ordering is not important), although it does have a slight performance cost.

8.1 Abstracting Persistence

In order to share as much code as possible, we abstracted the interface to the persistence mechanism through an interface called `Store`. This interface provides methods to set/get the root object, begin/end a transaction, and other miscellaneous methods needed to interface with the persistence mechanism. Mostly to aid the JOS and JBP implementations, the transaction methods support the notion of a read-only transaction, which applies to most of the OO7 operations. Not all of the persistence mechanisms actually require the use of transaction methods but including them maximizes code reuse.

There is a separate implementation of the `Store` interface for each persistence mechanism, named `PMStore`, where `PM` denotes the mechanism. For example, the JOS implementation is called `JOSStore`. When the benchmark is run, the choice of persistence mechanism is indicated by a command line option and the given store implementation class is loaded using the dynamic class-loading mechanism of the Java platform.

8.2 Class Structure and Inheritance

Each kind of object in the OO7 specification is modeled as a class. In the initial version, attributes were represented as fields that had public access, thus violating encapsulation. We changed

the design to make such fields private and introduced get/set methods as appropriate in accordance with standard conventions for the Java platform. This was most significant for the EJB and JBP variants, since these mandate that clients only access attributes through get/set method calls on the interface.

OO7 specifies some attributes, common to several types, that can be modeled by inheritance. For example, most OO7 objects include a "type" and a "build date". Also, base-assemblies and complex-assemblies both include an attribute that refers to the parent assembly, that can be factored into an "assembly" superclass. We chose to use class inheritance to model these attributes. The EJB variant required us to use an interface for this purpose, rather than a class, However, this variant required sufficient additional modifications that we did not deem it worthwhile to backport the use of interfaces to the other variants.

The root class that denotes the database is called OO7 and contains attributes recording the particular configuration and the Module instances that contain the data. The singleton instance of the OO7 class is all that need be accessed to display the configuration of the OO7 database, and this is one of the options of the driver module.

9. The Variants of OO7J

9.1 The Transient (VM) Variant

As a baseline test, we first developed a version of the benchmark that operates entirely in transient memory. The store implemention for this variant is called VM and is essentially a null implementation. In this variant, since there is no persistence, the database is always built first and then operated on.

This variant serves to set a baseline performance and scalability measure, and can also be viewed as a single-user main-memory database.

9.2 The OPJ Variant

No changes are required to the OO7 classes for use with OPJ. The store class, named PEVMStore, to reflect the particular OPJ implementation that we used, simply registers itself as a root of persistence in a static initializer and uses a static variable to store the OO7 root object. The commit of a write transaction is implemented by invoking a checkpoint operation.

9.3 The JOS Variant

The JOSStore implementation uses the JOS readObject and writeObject methods to read and write the serialized form of the OO7 database to a file, respectively. The name of the file is passed in as a JVM property and is accessed using the System.getProperty method, to avoid having to make a special variant of the OO7 driver program.

To avoid unnecessary output, JOSStore takes advantage of the read-only transaction information and only invokes writeObject when an update transaction commits.

All classes that are included in the OO7 data structure must be annotated to implement the java.io.Serializable interface. To avoid creating a variant for such a trivial change, this modification was pushed back into the VM variant, as it otherwise has no semantic impact.

9.4 The JBP Variant

The JBPStore implementation is similar to JOSStore but uses the XMLDecoder and XMLEncoder classes to read and write the persistent data, respectively. Initially it seemed simplest to modify the OO7 persistent classes to make them completely compliant with the JavaBeans model, that is, a zero-argument constructor and get/set methods for all the attributes. Most of the get methods were already in place due to the earlier decision to encapsulate data inside the classes. However, the set methods did not exist in the initial OO7J implementation as updates to objects are fully encapsulated in the traverse method.

With these changes we were able to write out OO7 objects with JBP, but problems occurred when trying to read them back. The essence of the problem was a dependency between two attributes that had its roots in the cyclic nature of the OO7 data structures. The ordering of the attributes in the persistent representation was the cause of the problem which, experimentally, could be fixed by manually editing the file and reordering the attributes. To be fair, the JBP specification states that attribute values must be independent. The ultimate solution to this problem was to use the DefaultPersistenceDelegate class to explicitly control the ordering of attribute initialization. This required adding multi-argument constructors to several classes and associating instances of DefaultPersistenceDelegate for those classes with the XMLEncoder. This causes the special constructors to be invoked when recreating the object graph, ensuring the correct order of attribute initialization. Apart from the extra constructors, all of these changes were localized to the JBPStore class.

9.5 The JDBC Variant

This variant naturally required some major changes, and was essentially a complete rewrite except for the driver module.

The SQL Schema

The original OO7 paper provides a standard entity-relationship diagram for the OO7 classes, and it is straightforward to translate this into a SQL schema using, for example, the recommendations in [Date95].

There is a choice, however, on how to handle the class inheritance. The standard approach, which we chose to follow, places each element of the class hierarchy in a separate table that contains the attributes defined at that level. Subclasses refer to the parent table through a foreign key that also acts a the primary key in the subclass table.

For example, CompositePart inherits from DesignObj:

```
create table DesignObj (id integer, type varchar(10), build_date integer,
   constraint designObj_pk primary key (id)
);
create table CompositePart (id integer, rootpart integer,
   constraint compositePart_pk primary key (id),
   constraint compositePart_fk foreign key (id) references DesignObj(id)
);
```

In the design of the BUCKY benchmark [CDN+97] that also involved some of the OO7 designers, this approach was rejected in favor of collapsing the superclass attributes into the subclass, to avoid the table joins that are otherwise necessary to gather the complete set of attributes for a

subclass instance. In our case, we didn't think this was an important issue, although, as we shall see later, it turned out to be significant for other reasons.

The complete relational schema is included in Appendix B.

Writing the JDBC Variant

Writing the JDBC variant was mostly straightforward, and confirmed that JDBC is indeed easy to use. Initially, as seems be common practice, we did not go to the length of creating the table definitions via JDBC, but ran these directly under the SQL interpreter provided with the database. Later, when it became convenient to perform end-to-end runs under program control, we added code to drop and create the tables from the benchmark when creating the database. This was a very straightforward process. We tested the program on several databases but only report results for one.

The large text objects specified for OO7 caused some problems, since this is an area where databases differ in their support, even to the extent of using different names for the large text type. Moreover, we were unable to simply use the JDBC `setString` method as the size exceeded the JDBC driver limits. Fortunately, JDBC provides a different method, `setCharacterStream`, that allows the driver to send the data in chunks so we used this instead.

When implementing the navigational methods of OO7, we were faced with implementation choices and also hit some scaling problems that required some rewrites. The first traversal operation simply visits every object in the assembly structure and then every atomic part in its associated set of private composite parts. Since the atomic parts form a cyclic graph, it is necessary to record which parts have been visited to avoid infinite looping. The obvious way to implement this is as a table that contains already visited parts. There is a choice of storing this transient table in the database as a SQL table or in the Java application as a hash table. Initially, we implemented both forms. As might be expected, the SQL table was by far slower because of the extra domain crossing involved so we dropped this approach.

The scaling problem related to the recursive nature of the traversal operation. When implemented in the obvious way with a recursive method invocation, this results in one open database cursor for each level of the recursion, and the total typically exceeds the maximum open cursor limit of the database. This is usually configurable but requires a database restart. The workaround that we adopted was to read all the members of the result set, that is, all the children of a node, save them in a data structure in the Java programming language, and close the cursor before recursing.

Navigation is inherently slow because it requires a domain crossing for every navigation step. The OO7 queries on the other hand, could all be implemented in such a way that the majority of the work took place in the SQL domain, with no object construction required on the Java application side.

Since the traversals involve the repeated application of similar SQL statements, differing only in the arguments, it is advantageous to use the JDBC `PreparedStatement` class as this reduces compilation overhead in the SQL server.

As the JDBC variant is a stand-alone program, it can retain complete control over the database connection management and statement caching, in contrast to the EJB version. We open a single database connection and hold it open for the entire interaction with the database. Auto-commit is turned off, so the entire database creation or update is a single transaction. Also we explicitly cache `PreparedStatement` instances.

With hindsight, the JDBC variant is the odd one out in that it does not attempt to retain any semblance of object-orientation. In particular, the OO7J object graph of the VM variant is not con-

structed. A more accurate comparison, particularly with the JDO-TP and EJB variants, would have been achieved by constructing the OO7J object graph, with explicit reading and writing of field values from the database.

9.6 The JDO Variants

During this research, the JDO specification was evolving towards its first public release under the Java Community Process and initially there was no reference implementation available that could support OO7. However, an early access version that was based on the JDO model, was available in the Forte™ for Java™ development environment as the *transparent persistence* module (JDO-TP).[12] Later in the study, the first public release of the JDO specification occurred and a reference implementation (JDO-RI) was released with a store based on a simple btree-based file. These two implementations show one the strengths of JDO, namely the wide variety of different external stores that it can work with. We report results for both implementations below.

JDO 1.0 requires that all persistent classes have a public zero-argument constructor. Beyond that, no source changes were required to the persistent classes. To make the OO7 object graph persistent it suffices to call the `makePersistent` method of `PersistenceManager` on the root `OO7` object.

JDO-RI

The `JDORIStore` class encapsulates the use of the `PersistenceManager` class. As with JOS, the pathname to the file containing the persistent representation is passed in using a system property. We also had to create the XML file specifying which classes were persistent, that is used by the JDO-RI command line tool to perform the standard bytecode enhancement process, and also accessed at run-time to facilitate the persistence mechanism.

JDO-TP

Note first that JDO-TP is based on an earlier specification of JDO and so is not directly comparable to JDO-RI. JDO-TP uses object-relational mapping, supporting a variety of databases, and leverages a number of capabilities of the Forte for Java (FFJ) environment. JDO-TP can capture a database schema and save it as a file system object. This allows development to take place without a live database connection. The bytecode enhancement process is handled transparently by the FFJ environment. The bytecodes generated by JDO-TP are not compatible with JDO-RI due to the different versions of the JDO specification. Although JDO-TP is no longer a supported product, we believe that it is an accurate reflection of any similar system based on object-relational mapping.

JDO-TP can operate in two modes. In the first mode, it will generate new Java classes that correspond to the tables in the database. In the second mode, existing classes can be mapped by hand to the schema. JDO-TP infers relationships by analyzing the primary and foreign key structure of the schema. Since we already had an existing implementation of the classes for OO7, we used the second mode.

The schema that we imported into JDO-TP was the same one that we used with the JDBC variant. Since we had generated the schema initially by a straightforward transformation of the entity-relationship model, and modeled inheritance in the schema, we were optimistic that JDO-TP would have no difficulty mapping to it. We were hoping that we would be able to use exactly the

12 Support for this module has been dropped in recent releases. However, we believe it to be representative of JDO implementations based on object relational mapping.

same set of classes as the VM and OPJ variants. Unfortunately, the process was not quite so straightforward.

Inheritance

The first problem was that JDO-TP did not support inheritance.[13] It was unable to deduce and could not be told that the foreign key references to the DesignObj table indicated inheritance and that it should map the DesignObj attributes to the superclass fields. Fortunately, JDO-TP did allow multiple tables to be mapped to a single class and this could be used to map the DesignObj attributes. The price for this in OO7J, however, was that inheritance had to be removed and the DesignObj attributes included explicitly in the subclasses. This problem also affected the assembly classes, but more severely. In addition to having to collapse the inheritance structure on the OO7J side, we also had to collapse the tables in the relational schema. This was due to the use of polymorphism in the sub-assemblies set of a complex-assembly. This set is always homogeneous but may contain either base-assemblies or complex-assemblies, depending on what depth the complex-assembly is in the assembly hierarchy. However, the JDO-TP modeling tool requires a single specific class and table to be mapped for this set, and the only way to achieve that was to merge the subclasses into the superclass. So unlike the DesignObj case, where we eliminated the superclass, in this case we eliminated the subclasses and merged the attributes into the superclass.

One final mapping problem occurred. Each base-assembly contains two sets of composite-parts, the private set and the shared set. Since these form a many-to-many relationship, a join table is used in the schema. JDO-TP does support the use of join tables in such relationships. However, we had chosen to denote the private/shared attribute as an additional column of the join table. Again there was no way to inform JDO-TP that this field should be used as the discrimminant for determining which set was used to contain the relationship. The solution, which is in fact more object-oriented, was to create separate join tables, essentially moving the private/shared attribute into the schema.

By this stage, most of the code-sharing with the VM variant was lost, although, unlike the JDBC variant, it was only the structure and not the content of the classes that had to change. Some of these limitations would be lifted if JDO-TP fully supported JDO 1.0.

Object Identity

A second problem area was related to object-identity. JDO-TP required a separate class to denote the identity[14] of a persistent object. These classes are generated automatically, as nested classes, when the classes are created by JDO-TP, but not when mapping existing classes. There is no intrinsic need for these classes from the OO7J perspective; they are simply extra baggage. The simple solution was to use the generate mode to create the identity classes and then add them, as separate classes, into the OO7J package.

It turns out that the need for a separate JDO identity has impact beyond just requiring the extra classes. Although many of the OO7 classes do have unique integer fields that correspond directly to the primary key in the SQL schema, several of them do not. For example, the Connection class contains two object references to AtomicPart objects. Since duplicate connections are not allowed, these two references uniquely define the Connection object and therefore form a suitable primary key. It would be possible. therefore, in the VM or OPJ variant to maintain a hash table keyed on the values of this pair of references since the hashCode method is invariant.[15] In the

13 It should be noted that this is an JDO-TP implementation restriction and not a limitation of JDO.
14 In JDO terms, this is “application” identity.
15 OO7 does not actually require such a table.

JDO-TP version the rules for the identity class require that the integer fields from the database (corresponding to the foreign keys that denote the references) must also be stored as fields of the corresponding class. For the Connection class this requirement adds two, otherwise useless, fields.

The navigation operations execute exactly the same code as the VM variant, the relevant objects being instantiated automatically by the JDO-TP run-time as needed.

The final task was to determine if JDO-TP could create the database from scratch and not depend on the JDBC variant for this function. The support for persistence by reachability in the JDO model, meant that this should be achievable by executing the same code as the VM variant and then calling the makePersistent method on the root object (the OO7 singleton instance). Unfortunately, the OO7 data structure contains several circular structures, notably the ring of connected atomic parts, but also bi-directional relationships such as between a module and its associated manual. The version of JDO-TP we were using could not handle the first kind of circularity, but it could handle the second provided that the relationship was marked as a *managed* bi-directional relationship in the mapping tool. The workaround for the ring of atomic parts was to commit the transaction after all parts were created, but before the last connection in the ring was made. A new transaction was then started in which the ring was connected.

One orthogonality issue arose through the use of managed relationships to circumvent the circularity problems. The code of OO7J is written to explicitly manage relationships, that is, the code sets both sides of the relationship explicitly. When a relationship is managed by JDO-TP, code is generated automatically to handle the "other" side of the relationship as part of the enhancement process. For a 1-1 relationship the extra code is simply redundant - the only impact is on performance. However, for a 1-many relationship, such as the connections between atomic parts, attempting to add an element to a set twice will cause an exception to be raised. We had to modify the code to ignore this case, with the cost that we lose the ability to distinguish between an erroneous attempt to add the same element twice and the double add from the implicit management.

Arguably, the correct modification would be to remove the explicit relationship management from the code and rely on the implicit management by JDO-TP, although this would make the code persistence dependent. Unfortunately, there is a *catch 22* due to the fact that the implicit management only operates on objects that are known to be persistent. Before the first transaction commit, the atomic parts and connection objects are all considered transient,[16] so the implicit management does not occur, therefore removing the code is not possible. The fix for this would be to call makePersistent on such objects to inform JDO-TP ahead of the commit. Unfortunately, the code would then become dependent on the persistence mechanism.

Finally, we experienced problems with the document and manual datatypes. These require the use of the non-standard SQL datatypes such as LONG. Not only does support for these datatypes vary among relational databases, additional restrictions on size are sometimes imposed by JDBC drivers. JDO-TP did not support LONG datatypes at all, and we finessed this by reducing the size of the document and manual objects when creating the database. To avoid additional variants we decided to limit the size of these objects to 1KB in all tests.

9.7 The EJB Variant

The EJB variant, like the JDBC variant, required an almost complete rewrite, due to the requirements and constraints of the EJB programming model. Although it is now considered a bad practice, for performance reasons, we map the OO7J object model directly onto entity beans.

16 The JDO specification only considers a new object, obj, persistent if makePersistent(obj) is called, or if obj is reachable from a persistent object at transaction commit time.

As with the JDO/JDO-TP variant an underlying external datastore is required. The EJB model supports both bean-managed (BMP) and container-managed (CMP) persistence. BMP implementations typically use explicit JDBC programming and our implementation corresponds closely with the JDBC variant. CMP typically uses some form of object-relational mapping, with the database access code automatically generated by the container, similar to the JDO-TP variant.

The EJB model requires that all calls to a bean, including calls from other beans, go through an interface which must inherit either from javax.ejb.EJBObject or javax.ejb.EJBLocalObject depending on whether the bean will be accessed remotely or locally, respectively. One of the quirks of EJB is that the bean implementation class cannot implement the bean interface directly, as would normally be the case. Instead the container matches up methods by name and signature between the remote interface and the bean implementation class. In consequence, the developer loses the compiler check that all methods in the interface have been implemented. The suggested work-around for this is to create yet another interface, sometimes called the *business* interface, that does not inherit from EJB{Local}Object, and can be implemented by the bean implementation class, and then define the bean interface to inherit from both EJB{Local}Object and the business interface. We used the business interface approach to define the OO7J bean interfaces.

The first EJB variant of OO7J was coded to the EJB 1.1 specification, but only the BMP version was tested owing to the limitations of the object-relational mapping tools available at the time. Significant changes were made to the CMP specification between EJB 1.1 and EJB 2.x that required substantial changes. In 1.1, CMP attributes were specified as fields in the bean class. In 2.x, CMP attributes must be specified as abstract get/set methods in the JavaBeans idiom. The concrete bean class that implements these methods is generated either by the container's CMP system or, in the case that BMP is used, manually by the bean developer. A further complication is that the initialization of the bean must be split into two methods in the CMP bean class. Scalar fields, in particular the fields containing the primary key, must be initialized in the ejbCreate method. Reference fields, that is, container-managed relationship fields, must be initialized in the ejbPostCreate method.

The inheritance in the OO7 class hierarchy proved to be problematic in the EJB BMP variant, owing to the lack of multiple inheritance in the language. The CMP bean classes use single-inheritance, much like the VM variant. For example, the class BaseAssemblyCmp inherits from AssemblyCmp, which inherits from DesignObjCmp.[17] To code the BMP variant, each of these classes was subclassed again, for example, BaseAssemblyBmp extends BaseAssemblyCmp and contains the actual code to handle the manual JDBC-based persistence. The problem is that BaseAssemblyBmp needs to inherit from both BaseAssemblyCmp and AssemblyBmp and this is not possible due to the single inheritance limitation of the Java programming language. Our work-around is to create a separate AssemblyBmp object and store a reference to it in BaseAssemblyBmp and then use delegation to invoke its ejbCreate and ejbPostCreate methods.

The net result of conforming to the EJB specification is a rather complex set of interacting classes. It took quite a lot of debugging to get the application working correctly. Many of the problems were caused by the separation of the create code into the two methods.

We also ran into similar object mapping problems that we encountered in the JDO-TP variant, and solved them in a similar way.

It would be interesting to see how the experience would change if we started over and followed the current recommendations of using coarse grain EJBs. However, it is not entirely clear how to choose the grain and the benchmark operations do not provide much guidance. A CompositePart

17 We use Cmp as an identifying tag.

perhaps would make an appropriate coarse-grain object. However, although Connection objects are completely private, AtomicPart objects are returned as the results of queries.

10. Performance Measurements

One of the main goals in this study was to measure the impact of layered persistence mechanisms on application performance and scalability. The OO7 benchmark is unusual in not reporting a single performance number; instead the time to complete each operation is reported separately. Consistent with our goals, we are more concerned with the relative performance of each mechanism than with the absolute numbers.

The detailed performance data is collected together in Appendix A and referenced as appropriate in the discussion below. We present abstracts from the data inline in order to highlight comparisons between the persistence mechanisms.

The test environment consisted of two machines, one hosting the OO7J application and the other hosting the database server for those mechanisms that required a relational database. The OO7J machine was a dual-processor, 1015Mhz, SPARC® server with 4GB of main memory running Solaris™ 9 and the database server was a dual-processor, 450Mhz SPARC server with 512MB of main memory running Solaris 8. Locally attached disks were used for all database files. The physical separation of the database server is representative of a typical real-world multi-tier setup, but does add a small performance overhead compared to the persistence mechanisms running on the one machine. However, the performance of the SQL-based mechanisms is dominated by the SQL conversion and inter-process communication costs and not by the use of separate machines.

Virtual Machine

To maximize consistency, we use the same virtual machine, JDK 1.4.1_03, for all measurements, with the exception of the OPJ variant that uses a virtual machine derived from the Sun Microsystems™ Virtual Machine for Research (SRVM). We calibrated the relative performance of these two virtual machines by comparing their performance on the VM variant and found that they had essentially identical performance. Both of these virtual machines generate compiled code at run-time for frequently used methods. The critical OO7 methods are invoked enough times to trigger compilation in all runs except the initial run of the smaller database configurations.

The JOS and JBP variants required a non-default setting for the size of the Java stack frame in order to create the database and run the benchmark. JOS required a value of 1.5MB and JBP required a value of 4MB to handle all the database sizes. This is caused by the recursive depth-first traversal of the OO7 data structure that is used in the implementation of the JOS and JBP persistence mechanisms.

Garbage Collection

Performance measurements for Java applications can be very misleading due to the effects of garbage collection. Current Java virtual machines typically employ some form of generational garbage collection with fast young generation collections and relatively slow old generation collections. If an old generation collection happens to occur during a benchmark operation, the reported time can be affected significantly. Also, some of the persistence mechanisms make heavy use of the Java platform's weak reference mechanism for object caching, and this places addi-

tional strain on the garbage collector. The parameters controlling the dynamic compilation can also affect measurements. If runs are not sufficiently long, methods may not be compiled which can affect measurements by an order of magnitude. Fortunately, all the OO7 operations can be run multiple times in the same execution, thus limiting this effect to the initial, "cold" run.

Since the amount of heap memory can be varied and, indeed, may even have a different default for different virtual machines, there is a complex relationship between operation run-time and heap memory size.

Given the relatively high cost of a full garbage collection, it is clearly unsound to compare operation times unless the runs have similar garbage collection behavior. On the other hand, the amount of heap space required is a relevant measure of scalability, since the ultimate scalability of a mechanism is limited by the physical memory available to an application.[18] During testing we noticed some wide variations in the amount of garbage created with different persistence mechanisms and with different components of such mechanisms. For example, some JDBC drivers create much more garbage than others. Creating large amounts of garbage is not necessarily a problem, although it obviously reduces the time available to the benchmark proper unless there is sufficient memory to avoid any collections.

To minimize the impact of garbage collection, for each OO7 database configuration and persistence mechanism (except EJB), we first determined the heap size needed to create the database without any garbage collections, using a simple binary-chop process and the "verbose:gc" flag to the JVM. The benchmark runs are then made with a heap at least that large. To assess the memory footprint impact in a running system, we also measured the minimum heap size needed to create the database with garbage collection permitted and report that figure. We did not make these determinations for the EJB variant because a J2EE server contains many additional systems beyond the EJB containers. However, we ran the server with the maximum heap size and made sure that the bean pools were large enough to avoid any passivation.

Benchmark Configuration

As noted, the OO7 benchmark is highly configurable. The designers originally specified three main configurations: Small, Medium and Large.[19] These all share the same depth (7) and fanout (3) of the assembly hierarchy and the same number (500) of composite parts per module. Both the Small and the Medium databases consist of a single module; they differ in that the Medium configuration has 200 atomic parts per composite part, whereas Small has only 20. The Medium configuration also specified an order of magnitude increase in the size of the documents (20000 bytes) and manuals (1000000 bytes). The Large configuration is similar to the Medium, differing only in that it specifies 10 modules.

The original designers reported measurements for the Small and Medium configurations. Subsequent measurements in the literature have tended to report only on the Small configuration. In the decade that has elapsed since the original OO7 paper, both main memory and disk sizes have increased by an order of magnitude or more. This suggests that the Medium configuration would be the minimum configuration to use. Unfortunately, some of the studied persistence mechanisms have difficulty managing the Small configuration, let alone the Medium.

Since scalability is an important criteria, we chose to collect results across a range of configurations rather than report point results. There are three basic ways to scale an OO7 database:

1. Vary the number of Assembly levels.

18 Garbage collection and paging interact very badly and can introduce large performance penalties.
19 By modern standards, none of these configurations qualify as large.

2. Vary the number of CompositeParts.
3. Vary the number of AtomicParts.[20]

Options 2 and 3 scale the size of the database linearly. Options 1 and 3 scale the number of objects visited in traversals. The number of Assembly levels (L) determines the number of BaseAssembly objects, given by 3^{L-1}, each of which connects to 3 CompositeParts, thus giving 3^{L} total connections to CompositeParts. Ideally, a database contains more BaseAssembly connections than CompositeParts in order to maximize the probability that every CompositePart is visited during a traversal.

We chose the following two ranges:

1. **C20L4**: 4 Assembly levels, 20 CompositeParts and varying the number of AtomicParts in each CompositePart from 20 to 200 in steps of 20, thus providing a range of 400 to 4000 AtomicParts, 27 BaseAssemblies and 81 BaseAssembly to CompositePart connections.
2. **C200L6**: 6 Assembly levels, 200 CompositeParts, also varying the number of AtomicParts in each CompositePart from 20 to 200 in steps of 20 thus providing a range of 4000 to 40000 AtomicParts, 243 BaseAssemblies, and 729 BaseAssembly to CompositePart connections.

The total span encompassed by these two ranges starts below the Small configuration and ends close to the Medium configuration, but with less traversals owing to the reduction in the number of assembly levels.

Database Creation

All the persistence mechanisms could handle the C20L4 and C200L6 configurations, although both JBP and JDO-TP required numerous garbage collections at the larger range. Note that the JDBC variant creates no significant long-term data structures and so can create even the largest database in the minimum memory size, which we determined to be 1MB.[21]

Tables 6 and 7 show the minimum heap size in MB needed to construct the given database configuration for the C20L4 range and C200L6, respectively. Table 1 shows the average factor increase in heap memory usage relative to the VM variant. JBP is notably more prolific in its memory usage than JOS. Although the implementations of OPJ and JDO-RI are architecturally similar, the footprint of JDO-RI is higher then OPJ due to its store layer being implemented in the Java programming language, as opposed to OPJ which uses native code inside the JVM.

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP
C20L4	3.39	1	16.53	1	18.53	6.3
C200L6	2.56	1.31	19.19	0.21	20.6	8.19

Table 1: Average factor increase in memory size to create database relative to VM variant

The time to build the OO7 database is not typically reported, but for our purposes it is an important scalability measure, as it reveals the ability of the persistence mechanism to handle bulk-loads. Tables 8 and 9 show the time needed to construct the given database configurations.

The variation in performance is quite startling. Taking the VM variant as the baseline, there is a factor of nearly 400 between the fastest and slowest result in the C20L4 range and the factor is

20 Note that this also implictly scales the number of Connection objects, since there are 3 Connection objects for each AtomicPart.

21 The minimum memory size permitted by the JVM was 1MB.

even larger for the C200L6 range. Table 2 gives the average slowdown factor across the ranges relative to the VM variant.

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
C20L4	1.56	8.46	223.13	135.9	31.15	314.04	118.43	116.18
C200L6	2.34	10.65	831.41	327.17	42.51	1767.25	317.56	2532.34

Table 2: Average factor slowdown to create database relative to VM variant.

It seems evident that JBP has not been tested by its developers with large object graphs since its performance is very poor relative to JOS, much more so than can simply be explained by its use of XML rather than the custom binary data format of JOS. JDO-TP is also noticeably slower than JDBC and EJB, probably due to the fact that it was an untuned early-access version.

The relative performance between mechanisms is similar for the larger C200L6 range, but the relative slowdown increases for all the mechanisms. It is notable that the EJB-CMP variant is now twice as expensive as the BMP variant, suggesting a scaling problem in the CMP implementation. The extreme slowdown of JBP and JDO-TP can explained by excessive garbage collection activity.

Benchmark Operations

The OO7 benchmark operations are divided into navigational access (traversals) and queries. All traversals do a depth-first visit of the assembly structure, followed by an operation-specific visit of the components of the referenced CompositeParts.

We only report results for two of the OO7 traversal operations as there are a large number and, as noted subsequently by the designers [CDKN94], many of them do not provide significant additional insight. As noted earlier, we do not report results for the query operations since the query system, if any, is not one of the comparison criteria.

Each operation was run five consecutive times in the same run. The high and low values were discarded and the remaining three averaged to derive the reported result. However, we also report the highest value, since this corresponds to the "cold" run when data is being accessed from the store. Note that for the mechanisms that always read the entire database, i.e., JOS and JBP, the cold time is essentially independent of the data access pattern, as it is dominated by the time to read in the data.

Traversals

T1: Full Traversal

The basic traversal, T1, visits every AtomicPart in a CompositePart. Owing to the random connections between a BaseAssembly and its CompositeParts, some of the latter may be visited more than once during a traversal.

Tables 10 and 11 show the time in seconds for the cold T1 traversal for the C20L4 range and C200L6, respectively. Table 3 shows the average factor slowdown relative to the VM variant.

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
C20L4	0.99	19.53	303.91	102.24	81.4	560.96	499.47	384.15

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
C200L6	4.76	53.48	1718.75	322.41	131.42	2186.21	8946.5	16264.8

Table 3: Average factor increase for cold T1 traversal relative to VM variant.

The cold runs are generally dominated by the time to bring the data into memory. The slightly faster time for OPJ is explained by the fact that many of the virtual machine startup costs do not occur when executing from an existing persistent store, in particular the loading and initialization of classes. For the small datasizes of the C20L4 database range, the VM startup costs are significant in the total time for the run. The JBP overhead is very large relative to JOS, even exceeding JDBC. JDO-RI performs reasonably well in both ranges. However, JDO-TP and both EJB variants, scale very poorly especially in the C200L6 range.

Tables 12 and 13 show the time in seconds for the hot T1 traversal for the C20L4 range and C200L6, respectively. Table 4 shows the average factor slowdown relative to the VM variant.

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
C20L4	1.18	1.05	2.38	617.64	5.55	15.88	3123.64	583.92
C200L6	1.23	1.29	1.57	528.22	5.97	12.37	12281.27	8763.41

Table 4: Average factor increase for hot T1 traversal relative to VM variant.

What is especially notable in the hot run results is how the JDO caching works effectively to dramatically reduce the traversal times. EJB CMP performs significantly better than EJB BMP, but its scaling behavior is poor, the relative overhead increasing by an order of magnitude across the C20L4 range. OPJ shows the expected small percentage overhead and should be contrasted with JDO-RI which shows the larger overhead of bytecode enhancement versus custom compiled code.

T6: Sparse Traversal

The sparse traversal is similar to T1 except that only the root part of a CompositePart is visited. This should highlight persistence mechanisms that can perform efficient incremental access to persistent data. We only show results for the initial, cold, run of the benchmark, since hot runs only access cached data. All but JOS and JBP, the two mechanisms that have to read the entire database, showed a performance improvement, most by an order of magnitude. JDO-RI only improved by about a factor of two, which suggests that its store layer does not optimize incremental access well. One factor that can affect performance in an object store is how the objects are clustered in the persistent store and the granularity of data transfer. Table 5 shows the average factor increase for the cold traversal relative to the VM variant.

	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
C20L4	1.57	154.39	3796.43	125.27	374.43	427.49	520.75	63.98
C200L6	1.53	699.17	23073.33	205.83	837.68	744.27	3068.84	207.4

Table 5: Average factor increase for cold T6 traversal relative to VM variant

11.Conclusions

Access to persistent data is an important aspect of virtually all applications. The Java programming language follows the traditional approach of separating persistent data and transient data, leaving the management of persistence to applications or layered mechanisms. We reviewed the evolution of persistence mechanisms for the Java platform and proposed a set of criteria for comparison. A representative set of mechanisms was chosen for evaluation and each mechanism was compared qualitatively against the criteria. The mechanisms were then tested quantitatively by applying them to the implementation and execution of the OO7 benchmark.

All tests were run with sufficient memory that the databases were completely memory resident, as is common for many applications with today's large main memory sizes. The VM variant of OO7J set the performance baseline, against which the mechanisms were compared. The PJama implementation of OPJ scored best overall. It required the least source code changes, none to the main body of OO7J, and, while only a research prototype, it had by far the best performance. However, it did require a custom virtual machine. The EJB framework came out worst, requiring the largest number of changes to the source code and delivering the worst performance and scalability. JOS and JBP, while suffering from performance issues when reading and writing data due to their lack of incremental access, required minimal source changes and ran at full speed with data in memory. JDBC had a large impact on the source code but performed well given the large number of JVM–to–database boundary crossings. The two JDO variants demonstrated the versatility of the JDO model with respect to different external data stores. In particular, JDO-TP and its transparent object mapping worked well with a relational database, while still allowing the source code to stay very close to the baseline variant. Furthermore, the JDO performance overhead, once data was in memory, was reasonable, certainly compared to JDBC or EJB. Although EJB CMP did perform better than EJB BMP, it performed much worse than JDO-TP. The cause of the poor EJB performance is not clear, but it should be noted that some application servers had even worse performance. A thorough investigation into the reasons for all the performance problems would be interesting, but is beyond the scope of this study.

In conclusion, it is clear that the impact of persistence on application code covers a wide spectrum, as does the performance of the resulting system. OPJ has shown that it is possible to deliver very competitive performance if support is added at the VM level. At the other extreme, acceptable EJB performance seems unattainable at present unless dramatic changes are made to the application object model to avoid fine-grain objects when mapped to EJB. While this approach is now reflected in standard design patterns for EJB, the extra effort that it implies is disturbing from the overall application design perspective. In contrast, JDO manages to achieve reasonable performance at much lower impact on application design, while remaining agnostic to the nature of the external data store. Therefore, at this time, JDO would seem to offer the best overall persistence mechanism for demanding, object-oriented, applications. Note, however, that at the time of writing, a new specification for entity bean persistence [Sun04] was being proposed for EJB 3.0 that would bring it much closer to JDO in spirit.

12.Acknowledgments

Laurent Daynès implemented the initial version of OO7J. Kirill Kouklinski implemented the CMP EJB variant of OO7J. Grzegorz Czajkowski, Mikhail Dmitriev and Jeanie Treichel read drafts of this document and provided helpful feedback.

13.References

[ACM01] Alur, D., Crupi, J. and Malks, D. *Core J2EE Patterns – Best Practices and Design Strategies*. Sun Microsystems Press, 2001, ISBN 0-13-064884-1.

[AJ00] Atkinson, M. and Jordan, M. *A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform*. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-90, June 2000.

[AM95] Atkinson, M.P., and Morrison R. *Orthogonally Persistent Object Systems*. VLDB Journal, 4(3), pp319-401, 1995.

[FEB03] Fisher, M., Ellis, J., Bruce, J. *JDBC Tutorial and Reference, Third Edition*, Addison-Wesley, 2003.

[Ced96] Cattell, R. (ed). T*he Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, 1996.

[CDN93] Carey, M., DeWitt, D. and Naughton, J. *The OO7 Benchmark*. Proceedings of ACM SIGMOD Int. Conf. on Management of Data, pp12-21, Washington DC, 1993.

[CDKN94] Carey, M., DeWitt, D., Kant, C and Naughton, J. *A Status Report on the OO7 Benchmarking Effort.* Proceedings of ACM OOPSLA, pp414-426, Portland, OR, October 1994.

[CDN+97] Carey, M., DeWitt, D. Naughton, J., Asgarian, M, Brown, P., Gehrke, E. and Shah, D. *The BUCKY Object-Relational Benchmark.* Proceedings of 1997 ACM SIGMOD Intl. Conf. on Management of Data, Tucson, AZ, May 1997.

[CS92] Cattell, R. and Skeen, J. *Object operations benchmark.* ACM Transactions on Database Systems, 17(1), March 1992.

[Date95] Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, 1995, ISBN 0-201-54329-X.

[DM01] Dmitriev, M. *Safe Class and Data Evolution in Lomg-Lived Java Applications.* Sun Microsystems Laboratories Technical Report, SMLI TR-2001-98, August 2001.

[GJS96] Gosling, J., Joy, W. and Steele, G. *The Java language Specification*. Addison-Wesley, 1996, ISBN 0-201-63451-1.

[GJSB00] Gosling, J., Joy, W., Steele, G. and Bracha, G. *The Java Language Specification. 2nd Edition.* Addison-Wesley, 2000.

[JR03] Jordan, D. and Russell, C. *Java Data Objects.* O'Reilly, 2003, ISBN 0-596-00276-9.

[JA00] Jordan, M. and Atkinson, M. *Orthogonal Persistence for the Java Platform – Specification and Rationale*. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-94, December 2000.

[LMG00] Lewis, B., Mathiske, B. and Gafter, N. *Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine*. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-93, October 2000.

[Sun96] Sun Microsystems Inc. *Java Remote Method Invocation.*
http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html

[Sun99] Sun Microsystems, Inc. *Java Object Serialization Specification.*
http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html.

[Sun01] Sun Microsystems, Inc. Forte for Java, Release 3.0,
http://java.sun.com/features/2001/08/forte3.html

[Sun03a] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE)*.
http://java.sun.com/j2ee/index.jsp

[Sun03b] Sun Microsystems, Inc. *JSR-000012 Java™ Data Objects (JDO) Specification.*
http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html

[Sun04] Sun Microsystems, Inc. *JSR-220 Enterprise JavaBeans™ 3.0.*
http://jcp.org/aboutJava/communityprocess/edr/jsr220/index.html

[TNL95] Tiwary A., Narasayya N. and Levy H. *Evaluation of OO7 as a system and application benchmark*. OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance, Austin, TX, October 1995.

Appendix A: Detailed Measurements

Minimum heap size in MB to create database

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP
20	1	1.44	1	4.13	1	4.13	1
40	1	2.44	1	4.13	1	8.13	1
60	1	2.69	1	8.13	1	8.13	4.13
80	1	3.06	1	12.13	1	16.13	4.13
100	1	3.31	1	16.13	1	16.13	4.13
120	1	3.56	1	16.13	1	20.13	8.13
140	1	3.94	1	20.13	1	20.13	8.13
160	1	4.19	1	24.13	1	28.13	8.13
180	1	4.44	1	28.13	1	32.13	12.13
200	1	4.81	1	32.13	1	32.13	12.13

Table 6: C20L4 range databases

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP
20	1	6.69	1.38	32.13	1	36.13	16.13
40	4.13	9.69	4.13	68.13	1	68.13	28.13
60	4.13	12.81	8.13	96.13	1	112.13	40.13
80	8.13	15.69	8.13	128.13	1	136.13	52.13
100	8.13	18.94	12.13	160.13	1	160.13	68.13
120	12.13	21.81	12.13	192.13	1	228.13	80.13
140	12.13	24.81	16.13	224.13	1	248.13	92.13
160	16.13	27.69	20.13	260.13	1	264.13	104.13
180	16.13	30.69	24.13	292.13	1	292.13	116.13
200	20.13	33.94	24.13	320.13	1	320.13	132.13

Table 7: C200L6 range databases

Time in seconds to create database

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.12	0.36	1.25	10.91	10.49	3.54	20.12	8.63	4.3
40	0.12	0.24	1.04	19.89	16.19	4.72	31.91	10.2	9.7
60	0.16	0.27	1.28	26.81	19.32	5.5	45.73	17.92	13.62
80	0.19	0.28	1.95	39.29	29.11	6.12	57.65	22.85	17.28
100	0.23	0.32	2.29	47.19	31.78	7.06	72.23	26.57	23.65
120	0.25	0.34	2.31	58.51	36.96	7.94	84.46	37.15	33.26
140	0.28	0.35	2.05	74.15	41.59	8.61	99.15	37.54	40.51
160	0.32	0.38	2.19	83.85	42.15	9.09	114.76	41.88	53.22
180	0.33	0.4	2.47	110.8	49.26	9.71	129.58	44.71	54.26
200	0.41	0.43	2.6	119.49	59.05	10.07	143.94	52.32	62.92

Table 8: C20L4 range databases

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.42	0.5	3.35	115.44	59.28	10.76	181.54	61.64	80.93
40	0.61	0.81	4.31	227.01	114.33	16.86	361.4	113.97	292.45
60	0.71	1.2	5.79	346.19	158.07	22.6	614.88	166.49	542.37
80	0.81	1.56	7.68	493.02	219.21	28.85	895.72	217.53	900.11
100	0.84	1.94	8.68	637.65	279.14	34.43	1244.27	250.93	1375.79
120	0.93	2.28	10.14	843.51	321.38	41.25	1665.74	323.88	3063.01
140	0.99	2.66	11.5	976.73	384.14	46.99	2110.37	355.92	3639.47
160	1.01	2.99	12.88	1101.18	442.32	52.74	2715.82	406.66	5133.32
180	1.03	3.44	14.2	1396.88	471.08	59.44	3138.9	456.22	6722.76
200	1.07	3.79	15.33	1569.02	520.33	65.58	3770.76	518.54	7936.28

Table 9: C200L6 range databases

Time in seconds for cold T1 Traversal

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.02	0.04	0.77	5.21	3.79	4.16	17.77	12.76	5.47
40	0.05	0.05	1.14	10.15	5.71	5.3	28.81	22.03	11.38
60	0.07	0.07	1.43	15.43	7.03	6.09	38.5	30.15	17.8
80	0.1	0.08	1.73	24.69	8.96	7.17	48.12	39.22	26.29
100	0.12	0.1	2.02	31.04	10.1	7.66	57.4	50.26	36.29
120	0.14	0.12	2.18	37.73	11.69	8.56	67.24	60.94	48.05
140	0.18	0.14	2.44	52.93	13.42	9.1	77.74	72.31	63.18
160	0.17	0.15	2.65	55.5	14.71	9.46	85.78	86.06	80.18
180	0.17	0.17	2.84	72.85	16.38	9.96	95.14	101.25	98.2
200	0.15	0.19	3.1	78.99	17.29	10.43	105.34	113.52	115.43

Table 10: C20L4 range databases

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.16	0.2	3.28	77.75	17.68	10.73	111.37	127.6	113.65
40	0.14	0.39	5.4	149.39	31.24	15.6	207.18	337.62	436.18
60	0.16	0.59	7.5	232.01	44.25	19.53	293.42	666.41	1029.05
80	0.16	0.8	9.6	325.92	56.94	24.08	387.47	1080.13	1845.4
100	0.21	1.02	11.64	386.07	70.18	28.29	482.62	1588.22	2853.9
120	0.22	1.26	13.74	529.69	83.22	32.67	564.66	2212.62	4015.28
140	0.25	1.47	15.62	623.51	97.25	37.07	659.02	3022.15	5609.76
160	0.28	1.74	18.01	709.04	108.79	41.6	749.34	3828.72	7496.31
180	0.31	1.96	19.72	938.23	122.67	45.75	842.9	4838.35	8768.91
200	0.34	2.18	22.27	*	136.73	50.43	929.99	5804.06	11154.1

Table 11: C200L6 range databases

* Could not complete due to java.lang.OutOfMemory error.

Time in seconds for hot T1 traversal

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0	0	0	0	1.88	0.01	0.03	6.63	0.33
40	0.01	0.01	0.01	0.01	3.32	0.03	0.07	12.5	0.8
60	0.01	0.01	0.01	0.02	4.7	0.04	0.11	19.61	1.73
80	0.01	0.01	0.01	0.02	6.14	0.05	0.16	27.84	3.19
100	0.01	0.02	0.01	0.03	7.37	0.07	0.19	36.6	5.64
120	0.01	0.02	0.02	0.05	8.86	0.08	0.24	46.58	8.82
140	0.02	0.02	0.02	0.06	10.46	0.1	0.29	57.71	12.85
160	0.02	0.02	0.02	0.03	11.68	0.12	0.34	69.63	16.97
180	0.02	0.03	0.02	0.08	13.19	0.14	0.39	84.09	24.36
200	0.02	0.03	0.02	0.04	14.61	0.15	0.43	96.83	28.12

Table 12: C20L4 range databases.

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.02	0.03	0.02	0.03	14.79	0.14	0.39	111.22	29.64
40	0.04	0.06	0.05	0.08	28.29	0.29	0.56	311.44	164.31
60	0.07	0.09	0.09	0.11	41.33	0.45	0.98	630.99	482
80	0.1	0.13	0.13	0.16	53.88	0.61	1.14	1032.8	622.42
100	0.13	0.16	0.18	0.21	67.03	0.79	1.38	1537	1105.61
120	0.17	0.2	0.22	0.27	80.91	0.95	1.79	2155.13	1803.59
140	0.2	0.24	0.27	0.32	94.08	1.13	1.98	2931.34	2210.49
160	0.24	0.29	0.3	0.37	106.66	1.32	4.05**	3711.25	2593.49
180	0.27	0.32	0.35	0.78	118.57	1.49	2.5	4738.18	3428.95
200	0.31	0.36	0.41	*	133.15	1.66	2.77	5711.99	4764.85

Table 13: C200L6 range databases

*Could not complete due to java.lang.OutOfMemory error.

** Full garbage collection occurred during actual traversal operation.

Time in seconds for cold T6 (sparse) traversal

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.01	0.02	0.61	4.35	1.34	2.88	4.78	2.62	0.77
40	0.01	0.02	0.86	9.27	1.36	3.1	4.72	3.47	0.75
60	0.01	0.02	1.09	15.85	1.25	3.79	4.59	4.24	0.63
80	0.01	0.02	1.3	26.6	1.38	3.79	4.45	4.84	0.64
100	0.01	0.02	1.54	33.46	1.32	4.03	4.68	5.51	0.65
120	0.01	0.02	1.77	42.11	1.45	4.19	4.65	5.95	0.68
140	0.01	0.02	2.01	52.65	1.41	4.34	4.86	6.83	0.71
160	0.01	0.02	2.22	68.23	1.3	4.52	4.42	7.16	0.74
180	0.01	0.02	2.48	68.44	1.35	4.72	4.36	7.23	0.65
200	0.01	0.02	2.7	86.46	1.36	4.95	4.62	8.16	0.69

Table 14: C20L4 range databases

APC	VM	OPJ	JOS	JBP	JDBC	JDO-RI	JDO-TP	EJB BMP	EJB CMP
20	0.02	0.03	2.94	82.35	3.77	5.31	13.08	24.21	3.5
40	0.02	0.03	5.16	167.33	3.82	7.57	12.87	31.57	3.76
60	0.02	0.03	7.39	251.49	3.68	9.82	12.77	38.78	3.83
80	0.02	0.03	9.53	355.57	3.77	12.13	13.55	46.06	3.64
100	0.02	0.03	11.73	458.95	3.89	14.29	13.15	52.78	3.6
120	0.02	0.03	13.94	551.87	3.49	16.47	13.61	59.96	3.78
140	0.02	0.03	16.04	643.16	3.56	18.81	13.2	66.01	3.68
160	0.02	0.03	18.65	793.16	3.6	21.15	14.02	73.65	3.7
180	0.02	0.03	21.25	983	3.48	23.23	13.86	81.49	3.92
200	0.02	0.03	22.99	0	3.68	25.61	13.22	88.69	3.69

Table 15: C200L6 range databases

Appendix B: OO7 database schema in SQL

```
CREATE TABLE OO7
 (Name VARCHAR(20), NumAssmPerAssm INTEGER, NumCompPerAssm INTEGER, j`
  NumCompPerModule INTEGER, NumAssmLevels INTEGER, TotalModules INTEGER,
  NumAtomicPerComp INTEGER, NumConnPerAtomic INTEGER, TotalCompParts INTEGER,
  TotalAtomicParts INTEGER, DocumentSize INTEGER, ManualSize INTEGER,
  CONSTRAINT oo7_pk PRIMARY KEY (name)
 );

CREATE TABLE DesignObj
 (id INTEGER, build_date INTEGER, type VARCHAR(10),
  CONSTRAINT design_obj_pk PRIMARY KEY (id)
);

CREATE TABLE Document
 (cp_id INTEGER, title VARCHAR(40), text LONG,
  CONSTRAINT document_pk PRIMARY KEY (cp_id)
 );

CREATE TABLE CompositePart
 (id INTEGER, rootpart INTEGER,
  CONSTRAINT composite_part_pk PRIMARY KEY (id)
 );

CREATE TABLE AtomicPart
 (id INTEGER, x INTEGER, y INTEGER, doc_id INTEGER, part_of INTEGER,
  CONSTRAINT atomicpart_pk PRIMARY KEY (id)
 );

CREATE TABLE Connections
 (type VARCHAR(10), length INTEGER, from_id INTEGER, to_id INTEGER,
  CONSTRAINT connections_pk PRIMARY KEY (from_id, to_id)
 );
```

```
CREATE TABLE Manual
  (mod_id INTEGER, title VARCHAR(40), text LONG, textlen INTEGER,
   CONSTRAINT manual_pk PRIMARY KEY (mod_id)
  );

CREATE TABLE Assembly
  (id INTEGER, super_assembly INTEGER, modul INTEGER,
   CONSTRAINT assembly_pk PRIMARY KEY (id)
  );

CREATE TABLE BaseAssembly
  (id INTEGER,
   CONSTRAINT base_assembly_pk PRIMARY KEY (id)
  );

CREATE TABLE ComplexAssembly
  (id INTEGER,
   CONSTRAINT complex_assembly_pk PRIMARY KEY (id)
  );

CREATE TABLE Modules
  (id INTEGER, db VARCHAR(20), design_root INTEGER,
   CONSTRAINT module_pk PRIMARY KEY (id)
  );

CREATE TABLE BaseAssemblyComponents
  (base_assembly INTEGER, component INTEGER, priv_or_shared VARCHAR(7)
  );
```

ABOUT THE AUTHOR

Mick Jordan is a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include programming languages, programming environments, persistent object systems and systems software. He has a Ph.D. in Computer Science from the University of Cambridge, UK. He was a member of the team that designed and implemented the Modula-3 programming language. He was the principal investigator of the Forest project at Sun Labs that designed and developed orthogonal persistence for the Java platform. He is currently working on the Barcelona project in Sun Labs investigating the application of the Multi-tasking Java Virtual Machine to the J2EE environment.