David Jordan / david.jordan@ericsson.com

An object database for embedded environments

Embedded Java

POET SOFTWARE RECENTLY introduced a 100% pure-Java object database called Navajo. Navajo is targeted at the new market for embedded small Java devices. POET's goal was to develop a full-featured object database with a relatively small footprint; their primary jar file has a size of 300 K. Before the Java 2 Micro Edition (J2ME) was launched, Personal Java served as the Java virtual machine (JVM) for this market. Navajo was originally designed for Personal Java. It should also run in the J2ME Connected Device Configuration once it becomes available. Navajo supports both JDK 1.1 and 1.2.

The majority of applications being developed for small Java devices merely need a solution for persisting their Java object models. A Java application typically performs in-memory navigation and processing of these Java objects. Navajo facilitates this, because it maps objects between the database and memory on demand, with very little effort by the application.

In the small device market, often only a single JVM is running. Navajo is designed for storing information locally in the device. Navajo does not have a client–server architecture. With a single JVM in a device, it makes sense to have the database implemented directly on a `RandomAccessFile`. It is also very common for such devices to use flash memory for storage. POET addressed this by designing Navajo for the unique characteristics of flash memory, namely pagination and fast reads/slow writes, so it works well with the technology.

Navajo was developed while the Object Data Management Group (ODMG) was defining its 3.0 release, which has now been published. Navajo was released a month before the ODMG specification work was completed. The version described here is based on the initial release of Navajo, which is nearly compliant with the ODMG standard. I will note the exceptions. Navajo is likely to be upgraded to be 100% ODMG-compliant by the time you read this. In addition to support of the standard ODMG interfaces, POET has added many other useful features. Some are general database features, others are specific to the device market.

With Navajo, virtually any class can be declared to be *persistence-capable*. One of the features added in ODMG 3.0 is a syntax for a specification file to enumerate all the persistence characteristics of classes. This syntax was adopted near the completion of ODMG 3.0, so the Navajo version I evaluated still had its own proprietary specification file. Every class that the application persists in the database should be listed in this file. In addition, you can specify whether certain fields should be transient. Or, in the case of fields declared transient in the `.java` file, you can optionally indicate that they should be persistent in the specification file.

Supporting transparent persistence requires either a preprocessor, postprocessor, or special-purpose JVM. POET chose the preprocessor approach. The classes listed in the specification file must have their source code preprocessed. The preprocessor captures schema information and adds additional code and data to the class to enable transparent persistence. The preprocessor then invokes the standard Java compiler (javac) to compile to byte code. So, your build scripts would invoke POET's preprocessor instead of javac. POET also added support for this preprocessing to be seamlessly integrated into Borland JBuilder. Customer demand will likely drive its support of other IDEs.

Towards the completion of the ODMG 3.0, POET added support for explicitly making objects persistent and explicit deletion of objects. Prior to release 3.0, implementations were expected to support persistence-by-reachability and persistent garbage collection. What this means is that an object becomes persistent when it is referenced by another persistent object and remains persistent until there are no longer any references to the

David Jordan is with Ericsson in Research Triangle Park, NC. He specializes in object and database technologies, is the author of *C++ Object Databases*, and is the Java Editor of ODMG. David can be contacted at david.jordan@ericsson.com.

object in the database, at which point the object is garbage collected. Navajo does not support persistent garbage collection; however, it does support persistence-by-reachability. POET chose not to include persistent garbage collection because it would have increased Navajo's footprint beyond the market requirement of the small device market. Due to the late adoption of these features in ODMG, Navajo has a different syntax for them. But by the time you read this, they are likely to support the standard ODMG syntax.

Navajo supports an extensive set of collections. In addition to having collections that support all the ODMG interfaces, it has a collection for every collection interface defined in Sun's Collection API. These collections contain references to objects. In addition, Navajo has a few collections that specifically store String instances. See Table 1 for a listing of each Navajo collection and the standard interfaces it supports.

An *extent* in an object database is the mechanism used by an application to access all the instances of a class (or a class and its subclasses). Except for support of subclasses, this is similar to a table in a relational database. Navajo supports extents, even though the ODMG has not established a standard interface for extents.

Navajo also supports *multi-field indexes* on a class. An index is associated with a particular class. An application can iterate from the beginning entry of an index or establish a starting location in the index by providing a key with values for one or more of the indexed fields.

Navajo provides extensive transaction capabilities. Navajo transactions support the expected ACID transaction properties of atomicity, consistency, isolation, and durability. All four transaction isolation levels defined in SQL92 are supported. An application can begin, commit, checkpoint, and abort a transaction. A commit ends a transaction and removes Navajo's references to objects in the application cache. A checkpoint commits the current updates to the database, but implicitly begins a new transaction and preserves all the objects (and their locks) in the cache. There is also an option to downgrade write locks to read locks at a checkpoint to allow for a higher degree of concurrency.

Locks are implicitly acquired as the application accesses and modifies objects, but Navajo also supports explicit locking. All locking is performed on an object-basis. Navajo supports a pessimistic, strict, two-phase locking strategy. If a lock cannot be immediately granted to a transaction, the transaction waits for a given time-out period for the lock to be released. The time-out duration can be set by the application. When a lock is released, multiple transactions may be attempting to acquire the lock, yet only one can acquire the lock at a time. So each transaction makes a given number of retry attempts, the application can set a parameter to determine the number of retries.

It is also possible for an application to assign *priority levels* to transactions. Higher priority transactions have precedence over lower priority transactions in acquiring locks. A priority-significance value (again a settable parameter) defines the difference in priority required for a higher-priority transaction to abort a lower-priority transaction to acquire a lock. And naturally, if there are multiple transactions waiting when a lock is released, the highest priority transaction acquires the lock.

Navajo allows a JVM to have one or multiple concurrent transactions executing. Each executing transaction has its own cache of objects it manages in the JVM. This is necessary to preserve the isolation of the transactions. All possible combinations of threads and transactions are supported. The simplest application has one thread executing one transaction. One thread can participate in multiple transactions. Another alternative is to have multiple threads, each with its own transaction that it manages. Each thread can be handling a different "client," with Navajo handling all the concurrency issues. Finally, Navajo allows for multiple threads to share one or many transactions. This is the most complex thread-transaction application archi-

Table 1. **Navajo collections support for ODMG and Sun interfaces.**

Poet Navajo Collection	ODMG 3.0 Collection Interface	Sun Collection API Interface
ArrayOfObject	DArray	List
ArrayOfString	DArray (with String substituted for Object for element type)	List (with String substituted for Object for element type)
BagOfObject	DBag	Collection
ListOfObject	DList	List
ListOfString	DList (with String substituted for Object for element type)	List (with String substituted for Object for element type)
MapOfObjectToObject	DMap	Map
MapOfStringToObject	DMap (with String substituted for Object for key type)	Map (with String substituted for Object for key type)
SetOfObject	DSet	Set
SortedMapOfObjectToObject		SortedMap
SortedSetOfObject		SortedSet

tecture and requires the application to perform some of its own concurrency control. But as you can see, Navajo provides complete flexibility so that you can organize your application's thread architecture any way you wish.

Navajo also supports *nested transactions*. While a transaction is executing, the application can initiate a nested transaction. This nested transaction may commit or abort. If it commits, the objects changed by the nested transaction have their state associated with the containing transaction. If the transaction is aborted, then the objects in the containing transaction have the same state they had prior to the start of the nested transaction. No work is committed to the database itself until the outermost transaction commits successfully.

Navajo provides a number of features that allow the application to help govern the cache management policies used. The application can control the size of the cache. This is important in small device markets that only can provide a limited amount of cache space. Navajo can automatically swap some objects from the cache to storage if the application requires more objects than will fit in the cache. Several parameters are provided to control this behavior.

The fields of a persistence-capable class can be either persistent or transient. When objects are read into memory, transient fields are set to a null or zero value by default. If the application wants to initialize them to some other value, Navajo provides a `Constraints` interface that allows the application to perform some processing after an object has been read from the database and before an object is written or removed from the database.

Navajo also supports event notification, using a JavaBeans-compliant interface. Applications can get notification of:

1. Opening and closing of databases.
2. Beginning, committing, and aborting of transactions.
3. Reading and writing of persistent objects.
4. Other clients' updates, deletes, and concurrency operations on a specific object or all objects of a class.
5. Throwing of exceptions.
6. Application-defined events.

Standard `EventListener` and `EventObject` interfaces from `java.util` are used. Applications can use these facilities to be notified of various database-related activities as they occur.

Navajo is targeted for applications like mobile PDAs that may be managing Personal Information Management (PIM) data. This data often needs to be synchronized with application data found on other systems, such as the user's desktop system or corporate data stores. POET does not provide a complete end-to-end solution, that would seamlessly integrate and synchronize the disparate data stores. No standards exist in the industry for these facilities, nor are there any market-leading solutions. POET therefore decided to provide a toolkit supplying the underlying technology to enable synchronization. Two services are currently provided: XML import/export and a log service.

The XML service is based on SAX 1.0 (Simple API for XML), defined in package org.xml.sax. Navajo supports the export and import of objects using this facility. Several of the XML element tags are flagged with attributes. These attributes are used by the Navajo XML service to represent persistent object identifiers and references. An object element in XML representing a persistent object has an id attribute with the value of an object identifier for the object. Each field that represents a reference to another object contains a ref attribute that contains the object id for the referenced object. These are necessary for preserving the relationships among the objects being exported.

Importing is also supported, but it is slightly more complicated than exporting because several application scenarios exist that have to be treated differently relative to these id attributes. One may be importing new objects, in which case no work with identifiers is necessary. If you are reimporting an object that may have been exported and changed, it is necessary to match the identifiers in the XML document with those in the database. And if you are updating objects that have counterparts in other databases, with different identifier mechanisms, the application must handle the management of the mapping between the external identifiers and Navajo identifiers.

A Log service is provided to aid in synchronizing with other data sources. It is based on the notion of a *replica*, which is like a virtual (maybe real) remote database. The application can define as many replicas as necessary. The developer identifies which classes and interfaces should be associated with a replica; multiple replicas may involve the same class. Once the log service is started, all changes made to objects associated with at least one replica are logged. Facilities are provided to start the logging, add and remove replicas, and iterate through the log. Log entries are kept in chronological order. An entry in the log contains all the changes made by one committed transaction. The entry contains a time-stamp indicating when the transaction committed. It also contains three sets of object identifiers: a set indicating all objects inserted, a set of all objects modified, and a set of all objects deleted. An object identifier only occurs in one of these three sets in a single entry, but may occur multiple times in different entries. The object identifiers can be used to access the objects themselves (except for the deleted objects). The application can iterate through the log for a given replica. The application can indicate whether it is done with a log entry while iterating through each entry in a replica. Log entries are shared across replicas. Navajo takes care of removing the log entry once all replicas that reference the log entry have been processed and the application has indicated the entry is no longer required.

Because Navajo is targeted for the embedded-device market, there are no database administration tasks that an end user needs to perform. The few administrative tasks that are necessary can be performed by an application via method calls, without requiring any end-user intervention.

The primary missing feature is a query language. But the footprint costs of supporting a general query language may outweigh the benefits, especially if the application mainly wants to operate on Java object models. The parsing, optimization, and execution of a general query language would add a significant footprint to Navajo. The all-Java SQL databases targeted for a similar market have a substantially larger footprint than Navajo. The SQL query language has its own elaborate data model that it must support, and this comes at a cost in footprint. So, it is not clear whether a general query language is worth the footprint costs it imposes when an application primarily wants to navigate their object models in Java. Much of the value-based lookups the application wants to perform can be done by using their index mechanisms.

For a relatively small footprint object database, Navajo provides extensive capabilities. If you are developing Java applications for the small device market and require persistence of your object models, I highly recommend that you investigate Navajo. ■

URL

POET Software Corp.
www.poet.com

A Wireless Effort

continued from page 52

phone directory's UI, neither usability nor intuition has been compromised.

Like the application launcher, the phone directory relies strictly on images to convey information to the user. These compact images are ideal for representing data in a small area, and their meaning is easily interpreted by even the most novice users.

Conclusion

The "Post-PC era" can be described by three words: connectivity, diversity, and speed. With all of the advantages it offers to consumers, the embedded market brings with it a new design philosophy that has forced companies to reevaluate how they deliver services. Product developers are burdened with the task of creating products that have a small price tag but that deliver high functionality. Furthermore, as a result of intense time-to-market pressures, they are expected to deliver these products faster than ever before.

Java is starting to prove itself as a feasible language for the embedded world, and as more embedded operating systems begin to support PersonalJava, the demand for applications to enable these devices will increase exponentially. ■

Dilshan De Silva is a director of product development and Susan Pearson is a technical writer at Espial in Ottawa, Canada. They can be contacted at ddesilva@espial.com and spearson@espial.com, respectively.