JAVA SUPPORT

Introduction

The J-Checkpoint component of the Database Rely suite enables applications written in the Java™ programming language to be restartable. J-Checkpoint is a framework for restartability for application developers who want to make Java programs restartable.

Minimum Requirements

J-Checkpoint requires a Java 2 runtime system, that is Java 1.2 or higher. Java 1.3 or high is recommended, because of its performance benefits for server-side applications. J-Checkpoint should also work under Java 1.2. No versions of Java under 1.2 are supported.

J-Checkpoint requires a database capable of supporting the BLOB datatype via its JDBC driver. J-Checkpoint has been tested with DB2 UDB (version 5 and above) and Oracle. J-Checkpoint supports Oracle 8i (i.e. 8.1.6) as the minimum version. We have not tested J-Checkpoint with any earlier version of Oracle's database. Other databases besides these may work, as long as they have JDBC drivers and underlying database support for streaming BLOB data into and out of BLOB columns in the checkpoint table. Exceptions include: Microsoft Access, which does not support BLOBs, and PostgreSQL, which does not support the streaming of BLOBs in its JDBC driver.

Note: Oracle apparently uses one internal cursor per JDBC statement/result set. It is easy to use up all the cursors allowed to the current Oracle instance by doing something such as calling the Checkpointer's `DumpCheckpointTable()` method in the transaction loop. The program will abend (but can be restarted!) with the error message "ORA-0100 maximum open cursors exceeded". For testing, it is possible to increase the number of cursors available by going to the Oracle instance's configuration file, `init.ora`, and adding the parameter open_cursors and setting it to a value higher than the number of iterations of the transaction loop. (Ex: "`open_cursors=1000`".) For normal checkpoint use, an internal cursor is created only on restart (the only operation that requires a result set), so cursor usage should be minimal.

Getting J-Checkpoint Running Quickly

Steps to get J-Checkpoint up and running:

1. Create the checkpoint table. J-Checkpoint uses the same checkpoint table as the regular product, so see the chapter on setting up the checkpoint library's checkpoint table.
2. Put J-Checkpoint's classes on your CLASSPATH for the Java interpreter.
3. Edit your code or the sample code provided to connect to your database.
4. Run a program.

J-Checkpoint is distributed as a Java JAR file which needs to be added to your application's CLASSPATH. Your distribution comes with sample programs (explained later) to demonstrate J-Checkpoint. (Be sure to put the password you received from SoftBase Systems in these samples before compiling.)

Add the file jc.jar to the CLASSPATH of the application. The package com.softbase.jcheckpoint contains the API classes. Applications need to import the J-Checkpoint package with a line similar to:

```
import com.softbase.jcheckpoint.*;
```

The Restart Framework

J-Checkpoint is an object-oriented framework for restarting an object-oriented program. Using the framework, applications can make their objects restartable.

J-Checkpoint automatically saves the state of all objects which participate in the *logical unit of work* (LUW). A transaction is made up of any number of objects which cooperate to perform whatever work is required to complete the transaction. Each object must have its state saved to the database, so in the event of a restart all of the objects can be restored to the state they were in at the time of the last checkpoint. Since the state information is saved to the same database that the transaction processing is using, the two are always synchronized.

The object-oriented framework extends the traditional restartability found in third-generation languages (such as C and COBOL). Restartable programs in these languages were centered on synchronizing the internal state of the logical unit of work (file positions, internal storage, etc.) with the COMMIT logic. The framework makes this paradigm more general, and involves capturing the state of all objects taking part in completing the current transaction, regardless of what function they perform. While repositioning files is perhaps less important in the object-oriented world, the integrity of the database is significantly more important because downtime for the database has a bigger impact. A "batch" Java program may be updating a live database which online users are accessing, and the potential for an abend to create problems is much higher than in the traditional batch world. Thus, checkpoint/restart in the new online world is as important as it ever was in the old "batch" world. (And, it is still possible to do traditional file repositioning with J-Checkpoint. See the example program for details.)

The steps in designing a restartable program which uses the J-Checkpoint framework are:

- Identify the logical unit of work (LUW). That is, the transaction which the program is performing. A restartable program must have some transaction (or iterative processing) at its core.
- Identify the objects which participate in the LUW. These objects will have to be made restartable.
- Identify the data which describes the object's state for each object. This data will have to be stored so it can be restored on restart.
- Implement the JCRestore interface to store that data. Details on how to do this follow.

A restartable program has five main phases:

- Setup: The program begins its execution and creates all the objects which will be restartable. It also opens the JDBC connection to the database. The application by definition is transaction based, so it turns off the autocommit option of the JDBC connection.
- Init: The application creates a Checkpointer object and passes it the Checkpoint ID for this job and the JDBC connection. The program registers all restartable objects with the Checkpointer. Finally, the program calls the Checkpointer's Init() method, which creates the original snapshot of the state of the running program in the database. A COMMIT is taken.

- The LUW: Usually as part of a loop, the program performs each logical unit of work, and at the end of each LUW, a checkpoint is taken and a COMMIT issued to the database.
- Term: After all LUWs are complete, the application calls the `Term()` method to stop the Checkpointer. This vital step in the process erases all of the Checkpointer's internal data which was placed into the Checkpoint table so no restart is possible. A final COMMIT is taken.
- Cleanup: The application then cleans up any open JDBC connections, etc.

Important Notes

The Checkpointer object's `isRestart()` method should be called to see if the current run is a first run or a restart. If it is a restart, the program should take any actions its logic needs to prepare for a restart.

Calls to the Checkpointer object's `DoCheckpoint()` method and SQL COMMITs must be done together. We recommend having the Checkpointer control commits by calling the `DoCommits()` method with a boolean "true" value.

Rollbacks executed by the application program require that the application program rewind sequential file positions back to the point of the rollback. Corrections and adjustments must be handled by the application.

Programs that abend because of logic errors can be recompiled and then rerun as long as the objects which participate in the transaction loop do not change at all. The length, number, and definition of storage must remain the same.

To redo a job from the beginning without restarting it, it is necessary to manually delete the information about a checkpoint ID from the `cckpt_data` table. This is the format for the SQL statement to remove a job from the checkpoint table, where "myckptid" is the checkpoint ID for the job. The SQL statement would be:

```
delete from cckpt_data where ckptid='myckptid'
```

Is J-Checkpoint thread safe? Transaction processing itself is not thread-safe. All objects registered with a Checkpointer need to be modified only within a single transaction.

Making An Object Restartable

To make an object restartable, it needs to implement the standard Java interface `Serializable`. This allows J-Checkpoint to covert the object to a stream in order to save it to the database. This is the only change that must be made directly to an object. In addition, the interface `JCRestore` must be implemented for the object, either by the object itself or by a separate, helper object. `JCRestore` exposes a method that allows an object to restore itself (or allows a helper object to restore it).

All objects registered with a Checkpointer need to exist for the duration of the transactions being processed, that is between `Init()` and `Term()`. It is important to realize that only objects may be made restartable. Something that isn't an object, such as data-only static member classes, can't be directly restartable. (In this case, the non-object will have to be handled in the same manner as opaque objects described later.)

Restartable vs. Serializable

J-Checkpoint uses the `Serializable` interface provided by the Java runtime system for simplicity and performance. Serializing objects is typically equated with persistent objects. Restartable objects are different from persistent objects, as this list highlights:

Restartability is associated with saving the state of objects to a database. Traditionally, persistence has been associated with saving an object's state to a file. This perception is changing, however, as the advantages of using a database to store persistent information are realized.

Checkpoints are done to objects by an external controller (the Checkpointer), while a persistent object saves its state when it decides to.

Restoration of a restartable object's state is done automatically upon restart, while traditionally a persistent object must restore itself.

A restartable object's lifetime is until the job completes, which may include more than one run of the application to which it belongs. Persistent objects exist over multiple runs of the same program, preserving the object's state.

The JCRestore Helper Interface

Each restartable object needs to either implement the `JCRestore` interface itself or have a second, helper object which implements it. Depending on the object, either approach can be appropriate.

The reason for using `JCRestore` is technical and involves the Java runtime architecture. In our C and assembler restart products, restoring an "object" (in the Standard C sense of something that exists in memory, not a Java object instance) is to use a pointer to the object and directly update it. Java does not allow this.

In a nutshell, Java's serialization support does not allow any way to un-serialize the stored state of an object to an existing, already constructed object *in situ*. The only way an object may be un-serialized is to a new object. In most cases, this architecture will not present difficulties, because the handle to the object is updated to reference the new object, and an old object (if it exists) is sent off to garbage collection. Checkpoint/restart presents a wrinkle, because more than one handle to the object to be restored (at least) exists at the same time. For any restartable object in the program, both the Checkpointer and the user code has a handle to this object, and both must stay in synch: if either changes its handle, the other does not know this has happened and keeps a handle to the old object. The `JCRestore` interface allows the two to stay in synch by giving the user code a way to take the un-serialized object and make the original object look just like it.

The Checkpointer object also needs help in restoring the object, because Java does not allow dynamic casting. The Checkpointer itself has no way of knowing what class a `Serializable` object is supposed to be, and must rely on a helper object which does in order to convert the `Serializable` object over to its proper class.

Another benefit of the `JCRestore` interface is granularity: when restarting a program, it is often undesirable to restore an entire object. Only certain parts of the object need to be restored. The `JCRestore` interface lets the user code have total control over how a restartable object is restored.

Restarting Opaque Objects

Sometimes an object simply can't directly be made restartable. An *opaque object*, in this context, is one which does not allow serialization, or for which serialization doesn't make any sense. A good