Introducción

La idea de este trabajo práctico es poder entender cómo es el funcionamiento de un "Recovery Manager" de una Base de Datos para dejar la base en un estado consistente ante un **crash**. El algoritmo utilizado es el conocido como *UNDO-Logging* y se implementa la posibilidad de agregar un checkpoint para así ante un crash no es necesario leer todo el archivo log nuevamente. De esta manera, cuando un log es muy grande, se evita leer todo ante cada crash, leyendo sólo lo pendiente.

El tipo de checkpoint utilizado es el No Quiescente, el cual fue presentado en clase.

Características de uso

Se dan de alta los Items a utilizar y el nombre de las transacciones que van a formar parte del caso a probar. Luego se va armando el log uniendo las transacciones con los items y se ingresan los valores valores correspondientes. Esto se realiza mediante el botón Agregar Registro de Log. Tomamos que ese valor del item que se agrega en el log representa el valor anterior que tenía previamente ese ítem. Esto es porque usamos el algoritmo de UNDO-Logging y es necesario para deshacer las transacciones incompletas.

Estos registros de log pueden ser ingresados en cualquier orden y de cualquier forma, pero a la hora de analizar el log se valida que dicho log sea correcto. Más adelante se detalla cuáles son las características que tomamos para que un log sea válido.

En el caso de que el log sea válido, se ejecuta el algoritmo de recuperación donde luego se detallan las transacciones incompletas que serán las transacciones a deshacer, los cambios a realizar en la base de datos, y los cambios a realizar en el log.

Características implementadas

Es una implementación en donde no usamos la conexión con una base de datos real. Como mencionamos anteriormente, es una implementación en donde vemos cómo se comportaría el Recovery Manager.

Checkpoint no quiescente

Para agregar la funcionalidad de chekpoint no quiescente fue necesario incorporar nuevas sub-clases de `RecoveryLogRecord`: `StartCheckpointLogRecord` y `EndCheckpointLogRecord`.

Además de la interfaz gráfica se modifico `EditLogRecordDialog` para incorporar las opciones de start y end checkpoint. Y se creó una nueva ventana `StartCheckpointDialog` para permitir la creación de checkpoints en la interfaz.

Validación

Para incorporar la validación al método: `RecoveryLog.validate()`. Se creó la clase `LogValidation` que representa el algoritmo de validación.

```
package ubadbtools.recoveryLogAnalyzer.common;

…

public class RecoveryLog

{

	…
	public ValidationResult validate()

	{

		return LogValidation.executeFor(this);

	}

}
```

Durante la validación se verifica que:

- El primer registro de una transacción debe ser START *tx*.
- No hay registros de START duplicados.
- No hay registros de UPDATE *tx* después de un registro COMMIT *tx* o ABORT *tx*.
- No hay registros de COMMIT duplicados.
- No hay registros de ABORT duplicados.
- No hay registros de COMMIT después de un ABORT y viceversa.
- Los registros de START CHECKPOINT referencian a transacciones activas (fueron iniciadas pero no se les hizo COMMIT o ABORT).
- No puede haber dos START CHECKPOINT sin un END CHECKPOINT en el medio.
- Si hay un END CHECKPOINT antes debe haber un START CHECKPOINT.
- Al hacer END CHECKPOINT, todas las transacciones activas en ese checkpoint deben haber hecho COMMIT.

Como resultado de la validación se genera una instancia de `ValidationResult`, que responde a los siguientes métodos:

`boolean isValidationPassed()`

Retorna true si el log pasó la validación satisfactoriamente.

`String getValidationMessage()`

Retorna el mensaje con el problema de validación (o un String vacio si no hay problemas).

UNDO-Logging

Para incorporar UNDO-Logging al método: `RecoveryLog.recoverFromCrash()`. Se creó la clase `UndoLogRecovery` que representa el algoritmo de recuperación:

```
package ubadbtools.recoveryLogAnalyzer.common;

…
```

```
public class RecoveryLog

{

      …
      public RecoveryResult recoverFromCrash()

      {

            return UndoLogRecovery.executeFor(this);

      }

}
```

Los pasos de recovery son:

1. Recolección de transacciones incompletas y punto en el log hasta donde aplicar el undo (método `collectIncompleteTransactionsAndStartRecordIndex()`).
2. Undo de las transacciones (método `undoIncompleteTransactions()`).
3. Creación de resultado con la información obtenida (método `createResult()`).

Recolección de transacciones incompletas y punto en el log hasta donde aplicar el undo

Se lee el log del registro más reciente al más antiguo en busca de transacciones incompletas. Si hay un START CHECKPOINT con su correspondiente END CHECKPOINT, se marca la posición donde se encuentra el START CHECKPOINT para indicar al algoritmo hasta donde hacer UNDO (`startRecordIndex`).

```
for (int i = endRecordIndex; i >= 0; i--) {
    RecoveryLogRecord record = records.get(i);

    if (record.isCommit() || record.isAbort()) {
        completed.add(record.getTransaction());
    } else if (record.isStart() && !completed.contains(record.getTransaction())) {
        incompleteTransactions.add(record.getTransaction());
        newLogRecords.addFirst(new AbortLogRecord(record.getTransaction()));
    } else if (record.isEndCheckpoint()) {
        endCheckpointFound = true;
    } else if (record.isStartCheckpoint()) {
        if (endCheckpointFound) {
            // habia un endcheckpoint y se encontro su correspondiente start...
            // ya no es necesario seguir procesando el log
            startRecordIndex = i;
            break;
        } else {
            // hay un start checkpoint pero no su correspondiente end...
            // es necesario corregirlo en el log
            newLogRecords.addLast(new EndCheckpointLogRecord());
        }
     }
}
```

Undo de las transacciones

Simplemente se leen los registros desde el final del log hasta el punto indicado por el paso previo del algoritmo, y se generan acciones de recuperación de la base de datos que se representan con instancias de la clase `DatabaseRecoveryAction`:

```
for (int i = endRecordIndex; i >= startRecordIndex; i--) {
    RecoveryLogRecord record = records.get(i);
    if (imcompleteTransactionIsAffectedBy(record)) {
        record.addRecoveryActionTo(recoveryActions, i);
    }
}
```

Creación del resultado con la información obtenida

El resultado de la recuperación es representado por instancias de `RecoveryResult`, que brindan la siguiente información:

- Las transacciones a deshacer: `getTransactionsToUndo()`
- Los registros a agregar en el log: `getNewLogRecords()`
- Las acciones que deben realizarse para regresar la base de datos a un estado consistente: `getDatabaseRecoveryActions()`

Además de esta información provee algunos métodos por conveniencia para obtener un reporte descriptivo:

- `getTransactionsToUndoDescription()`: descripción de las transacciones a deshacer.
- `getDatabaseChangesDescription()`: descripción de los cambios a realizar en la base de datos.
- `getLogChangesDescription()`: descripción de los registros de log que hay que agregar.

Testing

El testing se realizo utilizando el framework JUnit. En particular se utilizó una caracteristica de JUnit 4.x llamada "Theories".

Los casos de prueba fueron tomados en base al mail con casos enviado a la lista de alumnos de la materia. Lo que se hizo para incorporar los casos de prueba es:

- Se creo una clase que representa los "recursos" con información usada en los tests: `ubadbtools.recoveryLogAnalyzer.RecoveryLogTestResources`
- Esta clase de recursos brinda instancias de `RecoveryLogTestData`.

Cada instancia de `RecoveryLogTestData` parsea el archivo .txt envíado. Solo se hicieron algunas modificaciones minimas respecto a los archivos de prueba enviados por mail (como quitar algunos "." al final de la linea o convertir el encoding a UTF-8).

Una vez cargados los casos de prueba los tests utilizan "theories" para testear la implementación con cada caso, por ejemplo:

```
@RunWith(Theories.class)
public class LogValidationTest {
    @DataPoint public static final RecoveryLogTestData BIEN1 =
RecoveryLogTestResources.testDataNamed("Bien1");
    @DataPoint public static final RecoveryLogTestData BIEN2 =
RecoveryLogTestResources.testDataNamed("Bien2");
    @DataPoint public static final RecoveryLogTestData BIEN3 =
RecoveryLogTestResources.testDataNamed("Bien3");
    @DataPoint public static final RecoveryLogTestData BIEN4 =
RecoveryLogTestResources.testDataNamed("Bien4");
    @DataPoint public static final RecoveryLogTestData MAL1 =
RecoveryLogTestResources.testDataNamed("Mal1");

....
    @Theory public void rejectInvalidLogs(RecoveryLogTestData testLogData) {
        boolean validationPassed = validate(testLogData.getLog()).isValidationPassed();
        boolean expected = testLogData.isValidationPassed();

        assertThat(validationPassed, is(expected));
    }
```

Lo que hace El framework es ir ejecutando: `rejectInvalidLogs(BIEN1)`, `rejectInvalidLogs(BIEN2)`, etc.

Este esquema de test se uso en: `LogValidationTest` y `UndoLogRecoveryTest`.

Modificaciones adicionales

Además de las modificaciones indicadas como TODO en los archivos del trabajo práctico se hicieron algunas modificaciones adicionales al código cuyas razones se describen a continuación:

Clases en el paquete `ubadbtools.recoveryLogAnalyzer.gui.forms`

- Se cambio en encoding a UTF-8.
  Parte del desarrollo fue realizado en Linux y MacOSX, el encoding de Windows generaba que se vean mal los acentos en la UI. Para no tener problemas entre los encodings por default de cada plataforma (UTF-8 en Linux, MacRoman en OSX e ISO-8859-1 en Windows) se decidió codificar todos los archivos en UTF-8.
- Se cambiaron los manejadores de eventos de los botones de "Mouse Click" a "Action Performed".
  El problema es que "Mouse Click" solo detecta clicks del mouse (no cuando se activa el botón en general), y hace que la aplicación responda de forma incorrecta en MacOSX y en Linux. La forma correcta es usar el evento "Action Performed" que se invoca cuando se "ejecuta" el botón (ya sea por teclado, mouse u otro medio).

Modificaciones en la jerarquía de `RecoveryLogRecord`

- Se modifico la clase `RecoveryLogRecord`. En muchos casos era necesario saber con que subclase se estaba trabajando, por lo tanto se agregaron metodos para evitar usar `instanceof`:
    - Métodos `isXXX`: `isStart`, `isUpdate`, `isAbort`, etc.
    - `public void addRecoveryActionTo(List<DatabaseRecoveryAction> recoveryActions, int logRecordIndex).`

  Como se meciona en modificaciones a futuro una alternativa a los métodos "`isXX`" es usar el patrón visitor. En ese caso tambien se podría quitar el método `addRecoveryActionTo`.

Modificaciones al build

- Para automatizar la compilación y testing se utilizo la herramienta Apache Maven, y por lo tanto se cambio la estructura de carpetas para soportarla.

Compilación y ejecución

Requisitos: Maven 2.x, JDK 1.6

Paso 1: Establecer el entorno

- Instalar el JDK (carpeta JAVA_HOME), y maven (MAVEN_HOME).
- Agregar Maven y el JDK al Path:

  Windows:

  ```
  set PATH=%PATH%;%JAVA_HOME%\bin;%MAVEN_HOME%\bin
  ```

  Linux/OSX:

  ```
  export PATH=$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin
  ```

Paso 2: Ejecutar Maven

En todas las plataformas:

```
cd ubadbtools

mvn install
```

Maven se encarga de compilar, ejecutar los tests y generar un .JAR. (Inicialmente Maven necesita de una conexión a internet para bajar automáticamente las dependencias necesarias).

El .JAR generado es un JAR especial (llamado UberJar) que contiene todas las librerías necesarias. Para ejecutar la aplicación:

```
cd ubadbtools
```

```
cd target

java -jar ubadbtools-1.0.jar
```

Para editar el código en Netbeans y Eclipse:

NetBeans 6.7+ soporta proyectos Maven. Simplemente usar la opción de Open Project y seleccionar el directorio ubadbtools.

Para Eclipse hay dos maneras:

1. Usar el plug-in m2eclipse: http://m2eclipse.sonatype.org/
2. Usar Maven para generar el archivo de proyecto Eclipse:

   Configurar en Eclipse una variable de classpath: `M2_REPO` apuntando al repositorio de Maven (ubicado en `$HOME/.m2/repository` en Unix o `%USER_PROFILE%\.m2\repository` en Windows).

   Luego ejecutar:

   ```
   cd ubadbtools

   mvn eclipse:eclipse
   ```

   E importar el proyecto eclipse al workspace.

Posibles modificaciones a futuro

- Los reportes de resultado se generan usando métodos de `RecoveryResult`. A futuro estos métodos se pueden separar en un objeto que reciba una instancia de `RecoveryResult` y genere el reporte con diferentes formatos (eso se puede hacer en la versión actual, pero no fue necesario para el reporte).
- Sería conveniente que los registros de log sean "Visitables" (patrón Visitor), para evitar el uso de `instanceof` o métodos del estilo `isXXX`.
- El recovery requiere que el log sea válido. Actualmente no se genera una exception si se invoca `recoverFromCrash()` con un log que no pasa la validación.