DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

17 de Julio de 2013 — Bases de Datos

Integrante	LU	Correo electrónico
Agustina Ciraco	630/06	agusciraco@gmail.com
Nadia Heredia	589/08	heredianadia@gmail.com
Pablo Antonio	290/08	pabloa@gmail.com
Vanesa Stricker	159/09	vanesastricker@gmail.com

1. Introducción

En este trabajo práctico, analizaremos el impacto de las diferentes estrategias para manejar los buffer pools. Nos concentraremos en el módulo *Buffer Manager*.

Las diferentes estrategias que consideraremos son

- un solo buffer
- múltiples buffers (distribución de Oracle)

Veremos las ventajas y desventajas de cada uno según la situación.

1.1. Formato

En primera instancia, presentaremos una breve explicación del uso de los múltiples buffer, luego se explicarán ciertos detalles de implementación, así como también las decisiones tomadas durante el desarrollo.
Para los detalles de implementación, se muestran las secciones de código que se consideran relevantes.

Por último, daremos nuestras conclusiones.

2. Múltiples Buffers

En esta sección se explica el funcionamiento de los múltiples buffer y su finalidad.

Basamos nuestro análisis en la bibliografía mencionada en la sección correspondiente.

Puede pasar que diferentes bloques interfieran entre sí, por lo cual en Oracle, se decidió utilizar *múltiples buffers*. Estos múltiples buffers en general se adecuan a la mayoría de los sistemas. Oracle implementa *múltiples buffers* utilizando una división de buffers basada en patrones de acceso atipico [1]. El tamaño de cada pool es configurable, pero los pool en sí son fijos, y son los siguientes:

- **Keep pool** Utilizado para objetos que siempre deberían estar en memoria, ya que son muy frecuentemente usados.
- **Recycle pool** Utilizado para objetos poco usados que no se desea que interfieran con el resto. En general son datos que no van a ser accedidos frecuentemente, y no se quiere que estos ocupen lugar en el pool default.
- **Default pool** Utilizado para todo lo demás.

[1] http://www.exploreoracle.com/2009/04/02/keep-buffer-pool-and-recycle-buffer-pool

3. Detalles de Implementación - Ejercicio 2

Las funciones de este ejercicio tenían que ver con el catálogo. Tanto para `Catalog` como para `CatalogManagerImpl` se realizaron tests de unidad, estos se encuentran en `tests/components/catalogManager`.

4. getTableDescriptorByTableId

En esta parte, teníamos que implementar una función que dado un `tableId`, devolviera el `tableDescriptor` correspondiente.

Para esto, simplemente recorrimos la lista `tableDescriptors` del catálogo, fijándonos elemento a elemento si el id era el buscado, usando la función `idMatch`, que compara un `tableDescriptor` con un `tableId` y devuelve si hay un match por Id.

```
public TableDescriptor getTableDescriptorByTableId(TableId tableId)
{
    /* Recorrer la lista y obtener el que buscamos. */

    TableDescriptor result = null;

    for (TableDescriptor t : tableDescriptors)
    {
        if (idMatch(t, tableId))
        {
            result = t;
            break;
        }
    }

    return result;
}

private boolean idMatch(TableDescriptor t, TableId tableId)
{
    return t.getTableId().equals(tableId);
}
```

5. loadCatalog

En esta parte, teníamos que incorporar la posibilidad de levantar desde un archivo XML el catálogo de la base de datos.

Para esto, utilizamos la función `fromXML` de `XStream`, que lee un archivo XML y lo parsea, devolviendo un objeto. Este objeto lo casteamos a un catálogo que es lo que necesitamos.

```
public void loadCatalog() throws CatalogManagerException
{
    XStream xstream = new XStream();
    String filename = filePathPrefix + catalogFilePath;

    try
    {
        catalog = (Catalog) xstream.fromXML(new FileInputStream(filename));
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
```

6. Detalles de Implementación - Ejercicio 3

La implementación de este ejercicio abarca varias clases, se muestran a continuación las clases que agregamos y modificamos.

La implementación final del buffer con múltiples pools está realizada en la clase `MultipleBufferPool`, ubicada en `core/components/bufferManager/bufferPool/pools/multiple`.

El test de unidad se encuentra en `tests/components/bufferManager/bufferPool/pools/multiple`.

6.1. TableDescriptor

Agregamos un campo tableBuffer, que es un String representando el nombre del bufferPool al cual la tabla está asociada.

Al construirse un nuevo `TableDescriptor`, si no se provee dicho valor de tableBuffer, se toma como "Default"

```
public class TableDescriptor
{
    private TableId tableId;
    private String tableName;
    private String tablePath;
    private String tableBuffer;

    ...

    public TableDescriptor(TableId tableId, String tableName, String tablePath)
    {
        this(tableId, tableName, tablePath, "Default");
    }
}
```

6.2. PoolDescriptor

Teniendo en cuenta la idea de la clase `TableDescriptor`, creamos la clase `PoolDescriptor`, que almacena el nombre y el tamaño de un pool.

```
public class PoolDescriptor
{
    private String name;
    private Integer size;

    ...
}
```

6.3. Catalog

Agregamos una lista de `PoolDescriptor` a la ya existente lista de `TableDescriptor`. Agregamos un método que, dado el nombre de un Pool devuelve su tamaño, buscándolo en la lista `poolDesciptors`.

Tener la lista `poolDescriptors` permite que se puedan tener todos los buffers que se desee, diciendo para cada uno el nombre y el tamaño. Esto hace que la implementación sea configurable y se tenga en cuenta la existencia de otros buffer más allá de los propuestos por Oracle.

```
public class Catalog
{
    private List<PoolDescriptor> poolDescriptors;
    private List<TableDescriptor> tableDescriptors;

    ....

    public Integer getSizeOfPool(String name) throws Exception
    {
        for (PoolDescriptor p:poolDescriptors)
        {
            if (p.getName() == name)
            {
                return p.getSize();
            }
        }

        throw new Exception("Name does not exists");
    }
}
```

6.4. MultipleBufferPool

Para implementar múltiples buffers, tenemos un mapeo entre `TableId` y `SingleBufferPool`.

Creamos un constructor de `MultipleBufferPool` que toma un `Catalog`, y primero construye un `SingleBufferPool` por cada elemento en la lista `poolDescriptors`, usando la información de nombre y tamaño. Luego se recorre la lista `tableDescriptors` para generar las asociaciones entre cada `TableId` y `SingleBufferPool`.

```
public class MultipleBufferPool implements BufferPool
{
    private Map<TableId, SingleBufferPool> tableMap;

    ...

    public MultipleBufferPool(Catalog c)
    {
        Map<String, SingleBufferPool> bufferPools;
        bufferPools = new HashMap<String, SingleBufferPool>();

        for (PoolDescriptor p:c.getPoolDescriptors())
        {
            SingleBufferPool buf = new SingleBufferPool(p.getSize());
            bufferPools.put(p.getName(), buf);
        }

        tableMap = new HashMap<TableId, SingleBufferPool>();

        for (TableDescriptor t:c.getTableDescriptors())
        {
            tableMap.put(t.getTableId(), bufferPools.get(t.getTableBuffer()));
        }
    }
}
```

Cada vez que le llega un pedido a una instancia de `MultipleBufferPool`, se fija en este mapeo cuál es el `SingleBufferPool` a la tabla o página en cuestión. Una vez localizado, se le delega a este la funcionalidad requeridad.

Por ejemplo, para la función `addNewPage` se realiza lo siguiente

```
public BufferFrame addNewPage(Page page) throws BufferPoolException
{
    TableId tId = page.getPageId().getTableId();
    return getPoolBufferFor(tId).addNewPage(page);
}
```

```
private SingleBufferPool getPoolBufferFor(TableId tableId)
                        throws BufferPoolException
{
    SingleBufferPool result = tableMap.get(tableId);

    if (result != null)
        return result;

    throw new BufferPoolException("The table is not being managed by the pool");
}
```

7. Resultados

7.1. Trazas utilizadas

Las trazas que utilizamos fueron las generadas utilizando la clase `MainTraceGenerator.java` provista por la cátedra, y dos generadas por nosotros, estas son:

- **Random0** Se ejecuta dos veces una transacción sobre 100 bloques distintos en una tabla de 200 bloques.
- **Random1** Se realizan 1000 accesos random a una tabla de 200 bloques.

Dichas trazas pueden encontrarse en la carpeta `ubadb/generated`.

7.2. Generación de resultados

Modificamos la clase `MainEvaluator` para ofrecer la opción de evaluar una traza dada utilizando `SingleBufferPool` y `MultipleBufferPool`.

Creamos la clase `TpEvaluator`, ubicada en `core/external/bufferManagement`, que lo que hace es correr sets de trazas y evaluarlas usando `SingleBufferPool` y `MultipleBufferPool`

```
public class TpEvaluator
{
    public static void main(String[] args)
    {
        PageReplacementStrategy pageReplacementStrategy;
        pageReplacementStrategy = new FIFOReplacementStrategy();

        runBNLJTraces(pageReplacementStrategy);
        runFileScanTraces(pageReplacementStrategy);
        runIndexScanTraces(pageReplacementStrategy);
        runRandomTraces(pageReplacementStrategy);
    }

    ...
}
```

Cada función `runXTraces` crea una lista con los nombres de las trazas correspondientes y se lo pasa a la función `runTraces`. Por ejemplo, para el caso de `indexScan` se realiza lo siguiente

```
private static void runIndexScanTraces
    (PageReplacementStrategy pageReplacementStrategy)
{
    List<String> traceFileNames = new LinkedList<String>();

    /* Index Scan. */
    traceFileNames.add("generated/indexScanClustered-Product.trace");
    traceFileNames.add("generated/indexScanClustered-Sale.trace");
    traceFileNames.add("generated/indexScanUnclustered-Product.trace");
    traceFileNames.add("generated/indexScanUnclustered-Sale.trace");

    System.out.println("\nIndex Scan\n");
    runTraces(pageReplacementStrategy, traceFileNames);
}
```

`runTraces` toma una lista de nombres de trazas y para cada una llama a las funciones correspondientes dpara evaluarla usando Single y Multiple buffer pools.

```
private static void runTraces
    (PageReplacementStrategy pageReplacementStrategy, List<String> traceFileNames)
{
    for (String traceFileName: traceFileNames)
    {
        try
        {
            System.out.println("\nTrace: " + traceFileName);
            evaluateWithSingle(pageReplacementStrategy, traceFileName);
            evaluateWithMultiple(pageReplacementStrategy, traceFileName);
        }
        catch(Exception e)
        {
            System.out.println("FATAL ERROR (" + e.getMessage() + ")");
            e.printStackTrace();
        }
    }
}
```

`evaluateWithSingle` toma diferentes tamaños de buffer, y para cada uno, evalúa la traza, llamando a la función correspondiente de `MainEvaluator`.

`evaluateWithMultiple` lee diferentes catálogos, previamente creados, que puden encontrarse en `ubadb/catalogs`. Cada uno de ellos tiene los buffers `Default, Keep, Recycle` con tamaños que van variando.

Hicimos en cada caso que el tamaño de los 3 buffers fuera el mismo y que además se correspondiera con el del single. Por ejemplo, si el tamaño utilizado para el single buffer pool era 15, todos los buffers del multiple buffer pool también serán de tamaño 15. Para la asignación en buffer, pusimos una tabla en cada uno de los buffers.

Para cada caso, se evalúa la traza, llamando a la función correspondiente de `MainEvaluator` y se muestra el hit rate obtenido.

Para las trazas de `BNLJ` tuvimos en cuenta solamente tamaños de buffer que fueran grandes, porque si no teníamos un problema de que no nos alcanzaba la memoria para todos los pedidos. En el resto de los casos, usamos tanto los tamaños chicos como los grandes.

```
private static void evaluateWithSingle
    (PageReplacementStrategy pageReplacementStrategy, String traceFileName)
    throws InterruptedException, BufferManagerException, Exception
{
    /* Para BNLJ. */ //int sizes[] = {252, 300, 500, 700};
    int sizes[] = {15, 20, 60, 120, 252, 300, 500, 700};
    System.out.println("\nSingleBufferPool\n");

    for (int size : sizes)
    {
        System.out.print(size + " ");
        MainEvaluator.evaluateSingle(pageReplacementStrategy, traceFileName, size);
    }
}
```

```
private static void evaluateWithMultiple
    (PageReplacementStrategy pageReplacementStrategy, String traceFileName)
    throws InterruptedException, BufferManagerException, Exception
{
    System.out.println("\nMultipleBufferPool\n");

    for (int i = 0; i < 8; i++)
    /* Para BNLJ*/ // for (int i = 4; i < 8; i++)
    {
        String filename = "catalogs/catalogSize" + i + ".xml";
        CatalogManager catalogManager = new CatalogManagerImpl("", filename);
        catalogManager.loadCatalog();

        Catalog catalog = catalogManager.catalog();
        MainEvaluator.evaluateMultiple(pageReplacementStrategy,
                                traceFileName, catalogManager);
    }
}
```

Los resultados obtenidos pueden encontrarse en `ubadb/results`

7.3. Resultados obtenidos

Los casos en los que el touch touch fue 0 se obviaron ya que no aportaban para poder realizar comparaciones.

8. Conclusiones

En esta sección presentaremos nuestras observaciones sobre las comparaciones explicadas en la sección anterior.

9. Biblografía

- **Oracle Docs** Oracle Database Concepts
  http://docs.oracle.com/cd/ E11882_01/server.112/e25789/toc.htm
- **Oracle Docs** Oracle Database Performance Tuning Guide
  http://docs.oracle.com/cd/ E11882_01/server.112/e16638/toc.htm
- **Artículo** Principles of Database Buffer Management, Effelssberg & Haerder
- **Reference to Oracle Technologies** Keep Buffer Pool and Recycle Buffer Pool
  http://www.exploreoracle.com/2009/04/02/ keep-buffer-pool-and-recycle-buffer-pool/
- **Praetoriate Oracle Support** Create Multiple Data Buffer Pools
  http://www.praetoriate.com/ t_oracle_net_data_buffer_pools.htm