Manual de cómo diseñar y desarrollar un sistema de cache en aplicaciones JAVA

por Carmelo Navarro Serna

Índice

1. Introducción

El uso del cacheo de datos no automático (no lo proporciona ningún Framework o Middleware que se use, se ha de implementar como parte de la aplicación) es una práctica muy poco utilizada debido normalmente a la falta de tiempo en el desarrollo. Esto es un error ya que supone una mejora en el rendimiento de cualquier aplicación. El propósito de este manual es establecer conceptos para definir el cacheo de datos, hacer un repaso de cómo está la tendencia del cacheo de datos actualmente al desarrollar aplicaciones JAVA y como plantear el cacheo de datos para mejorar una aplicación.

Voy a hacer referencia a tecnologías JAVA pero los conceptos y técnicas son extrapolables a otras tecnologías como .NET.

Existen muchos factores que intervienen el tráfico de datos dentro de una aplicación:

- La finalidad de la aplicación.
- El planteamiento utilizado para el desarrollo (patrones, diseño, técnicas, etc..).
- Hardware que tiene que soportar.
- Tecnología o tecnologías utilizadas para la implementación.
- Etc..

Pero lo que sí que es seguro es que no hay forma más rápida de acceder a los datos que tenerlos en la memoria donde se esté ejecutando la aplicación.

Lo más común es que los datos que utiliza una aplicación se guarden en un sistema externo (como una base de datos) por muchos motivos:

- Acceso distribuido: La aplicación puede ser el gestor de los datos pero no el único usuario de los datos.
- Durabilidad: Los datos deben durar si se “apaga” la aplicación.
- Planteamiento: La mayoría de planteamientos para el diseño de una aplicación usan un sistema externo de almacenamiento de datos.
- Capacidad: Por ejemplo, el volumen de datos que se puede guardar en base de datos es mucho mayor que el que se puede guardar en la memoria RAM de un maquina.
- Acceso a la información: Las base de datos ofrecen una estructura que permite hacer consultas muy complejas.
- Etc…

Si una aplicación puede tener en memoria todos los datos, o lo más usado, y aparte puede seguir con el uso del sistema externo de almacenamiento de datos entonces usar un sistema de cache hará que la aplicación tenga un mejor rendimiento manteniendo todos los beneficios de tener un sistema externo de almacenamiento de datos.

En este tiempo de memorias RAM gigantes no es descabellado tener cosas en memoria, por lo que la cache SIEMPRE es una mejora.

La información se puede almacenar de muchas maneras en una aplicación pero voy a centrarme en aplicaciones que guardan la información en una base de datos relacional.

La cache siempre se debe poder “desactivar”; es decir se debe implementar un sistema de cache como una capa adicional de la aplicación, donde el uso de esta capa no afecte a la lógica de la aplicación. Es decir, la aplicación debe de funcionar igual uso o no cache.

Para poder entender los conceptos de este manual es necesario:

- Tener conocimientos de Java y de la filosofía multihilo de Java.
- Conocer los frameworks de Hibernate, Spring y AspecJ.
- Conocer la especificación de persistencia JPA.
- Estar familiarizado con patrones de diseño de clases para el acceso a datos como DAOs.

Cuando se utiliza código para indicar como se puede implementar una parte del sistema de cache este podrá ser muy básico y con mucho acoplamiento en algunas ocasiones pero es muy sencillo de entender y refleja de manera mas clara el planteamiento que se quiere mostrar.

2. Conceptos básicos

Siempre que se desarrolla una aplicación, se piensa en una equivalencia entre los datos que se usan y una tabla de un modelo relacional. Para cachear una aplicación hay que pensar que estas tablas son de uno de estos dos tipos:

- Tablas Tipo: Son las típicas tablas de tipo de “algo” (Cliente, Expediente, etc…). Estas tablas no suelen crecer más de su volumen inicial y se suele consultar un registro en concreto o todos a la vez. Estas tienen campos que hagan referencia a otras tablas y si lo hacen es algo muy concreto y es a otra tabla tipo.

- Tablas Negocio: Son tablas que soportan las unidades de información del negocio de la aplicación. Son tablas de que hacen referencia a otras tablas de negocio y a otras tablas tipo. La consulta a estas tablas es aleatoria completamente (cualquier numero de registros y por cualquier criterio). Los datos pueden crecer y decrecer de forma aleatoria.

Según el criterio de almacenado que se utilice para los datos de cache existen dos estrategias:

- Si todos los registros que están en base de datos se guardan en la cache de la aplicación entonces **la cache es completa**. En caso contrario es incompleta o no completa.

- Si cada uno de los registros de cache tienen toda la información que su equivalente de base de datos entonces **la cache es coherente**. Si no todos los datos del registro equivalente de base de datos está en el registro de la cache o lo están pero se pueden desfasar entonces es incoherente o no coherente.

El objetivo es siempre buscar cache completa y coherente. Si por razones de espacio en memoria no puede ser completa entonces se buscará una estrategia clásica (buscar en cache y si esta coges el registro pero si no vas a base de datos) aunque siempre intentando que los datos más usados estén en cache. Lo que sí que desaconsejo es utilizar datos incoherentes en cache, seguro que sea cual sea el caso cuesta más o menos lo mismo marcar un registro como incoherente que actualizarlo.

Una práctica muy buena es empezar a desarrollar la aplicación sin base de datos y todos los datos guardarlos en memoria (en cache). Una vez que este claro cómo se van a guardar en memoria los datos, como se accede a ellos, como se actualizan, etc.. (una vez que este claro como es la capa de cache de la aplicación) completarlo con una capa de acceso a base de datos donde se replican los datos que han pasado por la capa de cache.

Al trabajar siempre con base de datos se crea la tendencia del uso tradicional de manejo de datos procedentes de base de datos. Esta tendencia consiste en que cuando se necesita algún dato entonces se recupera de base de datos una copia de esos datos mapeados en objetos JAVA básicos (Pojos) que se adecuan al código de negocio que utiliza dichos Pojos. Al procesar la información, si se necesita guardar información entonces se mapea la información que se desea guardar en otros Pojos que se utilizan para crear alguna forma de guardar la información en base de datos (como por ejemplo una sentencia SQL insert). Se puede ver en el siguiente ejemplo:

```
public class FacturaDao {

      public void crearFactura (String dni,  int tipoBonificacion){

            Cliente cliente = clienteDao.buscarCliente(dni);

            Gasto gasto = gastoDato.buscarGastoCliente(dni);

            switch (tipoBonificacion) {
            case 0:
                  // no tiene bonificacion
                  break;
            case 1:
                  cliente.setSaldo(cliente.getSaldo() + 1000);
                  break;
            case 2:
                  cliente.setSaldo(cliente.getSaldo() + 2000);

                  break;
            }
```

```
if (cliente.getSaldo() > gasto.getTotal()){
                  Factura factura = new Factura();
                  factura.setCliente(cliente);
                  factura.setGasto(gasto);
                  guardar(factura);

            }

      }
……
```

En la función “crearFactura” se busca el cliente al que hay que crearle la factura y el gasto que le origina en su cuenta dicha factura. Dependiendo del tipo de bonificación que se aplique, el cliente puede ver incrementado su saldo. Si el saldo del cliente es mayor que el gasto que origina la factura entonces se crea la factura.

Esta forma de trabajar no es incorrecta pero al ser “tan usada” produce el efecto de que cueste mucho utilizar otras metodologías. Como por ejemplo, a la hora de usar JPA se puede tener el siguiente problema:

```
public class FacturaDao {

      public void crearFactura (String dni,  int tipoBonificacion){

            entityManager.getTransaction().begin();

            Cliente cliente = clienteDao.buscarCliente(dni);

            Gasto gasto = gastoDato.buscarGastoCliente(dni);

            switch (tipoBonificacion) {
            case 0:
                  // no tiene bonificacion
                  break;
            case 1:
                  cliente.setSaldo(cliente.getSaldo() + 1000);
                  break;
            case 2:
                  cliente.setSaldo(cliente.getSaldo() + 2000);

                  break;
            }

            if (cliente.getSaldo() > gasto.getTotal()){
                  Factura factura = new Factura();
                  factura.setCliente(cliente);
                  factura.setGasto(gasto);
                  entityManager.persist(factura);

            }
            entityManager.getTransaction().commit();
      }
……
```

Esto no se debe hacer a menos que el propósito sea modificar el campo saldo del cliente en base de datos

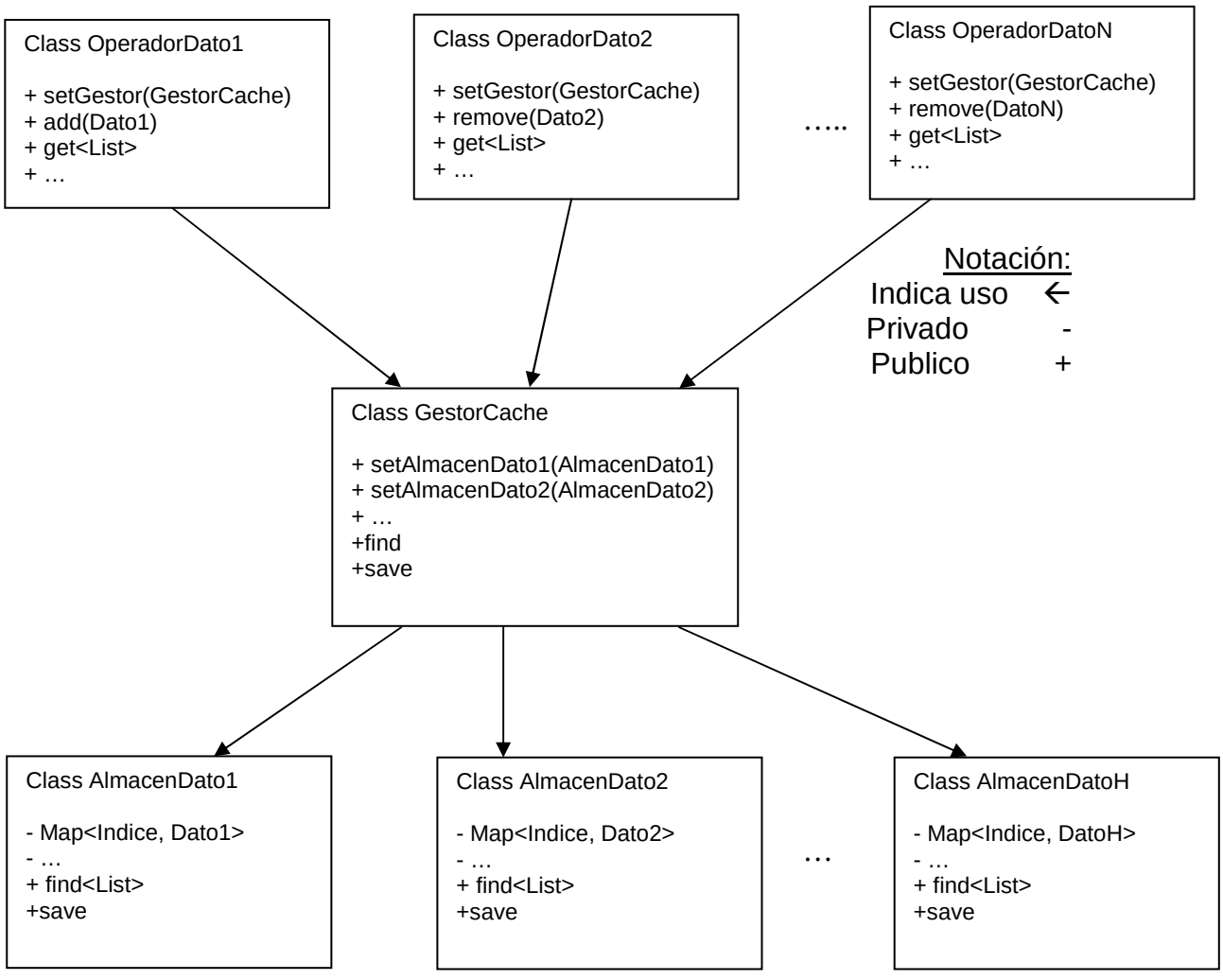
Esta es la forma más directa de “traducir” la manera tradicional de trabajar a usar JPA. Este código también crearía la factura pero al modificar la entidad cliente y dejar que se acabe la transacción se modificará el dato recuperado cuando no se quería guardar.

Este problema es parecido al que sucede si se usa un sistema de cache. Al usar cache, los datos están en memoria y una modificación al consultar el dato será una modificación en el dato origen. Se puede asumir y trabajar en consecuencia(es decir, saber que los datos consultados no se pueden modificar si no es ese el propósito) o hacer una copia de los datos cada vez que se consulte la información en cache. Implementaciones como EHCACHE te permite definir como hacer una copia de los datos.

3. Implementaciones

3.1 Tu propia implementación

La estructura de clases más aconsejable para implementar un sistema de cache es la siguiente:



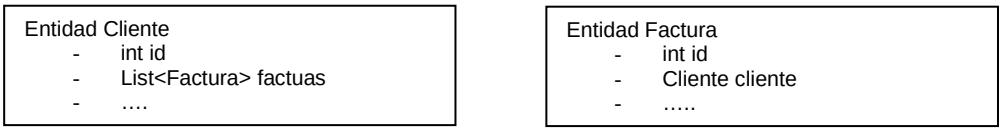
Hay tres tipos clases:

- Operador: La forma más utilizada por las implementaciones actuales de sistemas de cache (y por la implementación que recomienda este manual) es sustituir la llamada a una función pesada por otra equivalente que además de realizar la misma función accede al sistema de cache. Estas funciones son las típicas funciones que se pueden encontrar en clases DAO (getClientes, addCliente, etc..). Una clase de tipo Operador es la equivalencia a la clase DAO de la aplicación cuyos métodos pesados se quieren cachear en el sistema de cache.

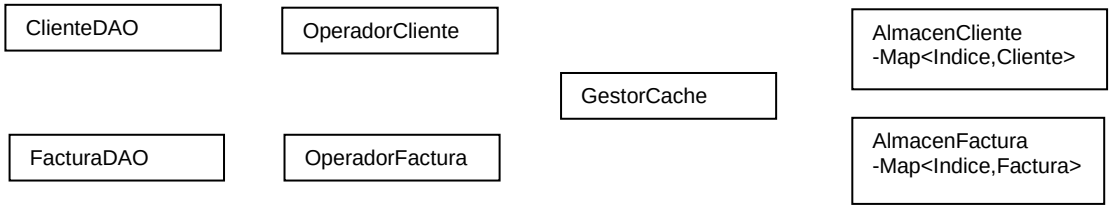
- Almacén: Son las clases donde se almacena la información. Tienen variables de tipo “Map” a nivel de clase donde se guarda la información a la que se accede por un índice y variables para el control de la concurrencia. También contiene operaciones simples sobre las variables de tipo “Map” como añadir un elemento, calcular índice, etc..Pero nunca consultas.

- Gestor: Esta clase es única y gestiona el uso de los almacenes por parte de los operadores. Esta clase no debe de tener lógica de la aplicación (toda se queda en los operadores) ni lógica de datos (toda se queda en los almacenes). Al estar toda la lógica en otras clases, los métodos de esta clase tienen que ser simples y cortos (es muy raro que tengas que utilizar una bucle para métodos de esta clase); esta clase es única porlo que si no se tiene cuidad puede llegar a tener excesivas líneas de código. Otra funcionalidad muy importante del gestor es el control de la concurrencia sobre los almacenes.

Siempre habrá igual número o menos de almacenes que de operadores, la mejor forma de ver el porqué es con un ejemplo. Supongamos que en una aplicación cliente se dispone de una clase ClienteDAO para las operaciones sobre la entidad Cliente y una clase FacturaDAO para las operaciones sobre la entidad Factura.

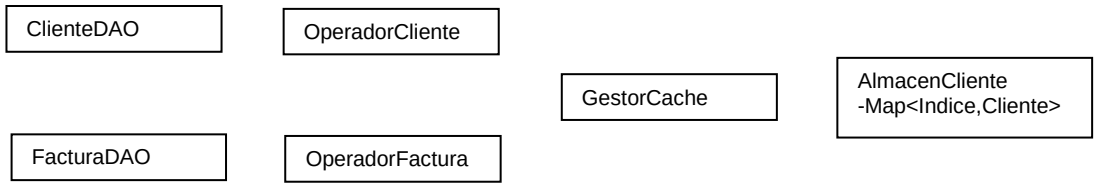


La forma más directa de implementar el sistema de cache es:



Pero si en la entidad Cliente hay una objeto List<Factura> que contiene todas las facturas de un cliente y es conocido que en la gran mayoría de consultas que hace la aplicación sobre la entidad Factura se accede por cliente entonces

lo más aconsejable es guardar las entidades Factura en los objetos List<Factura> de cada entidad Cliente. Por lo que la implementación del sistema de cache quedaría de la siguiente manera:



Si la aplicación tiene muchas consultas complicadas y además son las más utilizadas el sistema de cache puede volverse inmanejable. Siempre hay que intentar que a la información se acceda por un índice del almacén pero si no se puede entonces será necesario hacer operaciones de búsqueda como recorrer listas para encontrar los datos que se busca.

Siguiendo con el ejemplo anterior, si se busca la Factura cuyo campo ID es "A66C" y pertenece al Cliente cuyo Indice es 3 no se puede acceder directamente a la entidad Factura sino que habrá que buscar el Cliente en el objeto Map y después recorrer a lista de Facturas hasta encontrar la Factura cuyo ID es "A66C".

Cuando se utiliza una filosofía multihilo en la aplicación entonces se deben tener en cuenta el problema del acceso concurrente. Dos hilos de ejecución acceden a dos operaciones del Gestor, una de modificación y otra de consulta, intentan acceder al mismo tiempo al mismo almacén. Solución: Usar variables que se utilizaran como un flag y sirven para indicar si el almacén está siendo utilizado y sincronizar los almacenes en las operaciones del gestor. Un ejemplo de implementación seria:

```
public class Gestor {

    private AlmacenCliente almacenCliente;

    public Cliente buscarCliente (String dni){

        synchronized(almacenCliente){

            if (almacenCliente.estaOcupado())
                almacenCliente.wait();

            almacenCliente.marcarComoOcupado();
            ...

            // al terminar la operacion
           almacenCliente.marcarComoDesocupado();
           almacenCliente.notifyAll();
        }
        ...
    }
```

Código propio del Gestor

```
public void incluirCliente (Cliente cliente){

        synchronized(almacenCliente){

            if (almacenCliente.estaOcupado())
                almacenCliente.wait();

            almacenCliente.marcarComoOcupado();
            ...

            // al terminar la operacion
           almacenCliente.marcarComoDesocupado();
           almacenCliente.notifyAll();
        }
    }
    ...
}
```

Código propio del Gestor es completamente dependiente de cómo se desee implementar, es decir no hay una “receta” o patrón que pueda aportar este manual para indicar como se debe implementar. Lo que sí es muy importante destacar es que hay que optar por una buena política de creación de índices a la hora de guardar la información en los Almacenes.

Si no se va a tener una cache completa (todos los datos en memoria), el caso más acertado que una cadena compuesta por el nombre del método y el valor de los parámetros.

> Si se está cacheando la llamada al método buscarCliente (5) entonces solo habrá un Map y la clave seria “buscarCliente-5”.

Si por el contrario se va a tener una cache completa, entonces no hay que generar índices, se debe crear un Map en el almacén por cada índice. Es decir, si se dispone de la entidad Cliente y todas las búsquedas que hace la aplicación relacionadas con esta entidad son por id (un Integer) y/o por nombre (un String) se debe crear el siguiente Almacen:

```
class AlmacenCliente {

    private Map<String, Cliente> clientesPorNombre;

    private Map<Integer, Cliente> clientesPorId;

    private boolean ocupado;

    public boolean estaOcupado() {
        return ocupado;
    }

    public void marcarComoDesocupado() {
        ocupado = false;
    }

    void marcarComoOcupado() {
        ocupado = true;
    }
    ...
}
```

Los tipos de cache (completas e incompletas) se pueden combinar ya que se puede tener Almacenes con todos los datos de su tabla de base de datos correspondiente y otro Almacenes en los que, por el volumen de datos, solo se tengan los datos más usados.

> Para la implementación de las listas es mejor utilizar CopyOnWriteArrayList, penaliza mas la modificación pero evita los **ConcurrentModificationException.**

Ya se ha visto que es un Gestor, que es una Almacén y que es una clase Operador. ¿Pero cómo se implementa un Operador? ¿Cómo se sustituye la llamada a la función pesada de un DAO por otra más ligera sin afectar a la funcionalidad? Como ya se ha mencionado, las clases Operador sustituyen a las clases DAO de nuestra aplicación. La forma más elegante de cambiar DAOs por Operadores es crear una interfaz por cada clase DAO con todos los métodos que se desean ofrecer al resto de la aplicación y donde se usen DAOs en la aplicación cambiar la instancia del objeto por el Operador equivalente como en el siguiente ejemplo:

Se dispone de la siguiente interfaz:

```
public interface Datos {

    public Cliente obtenerClientePorDni(String dni);

    public void insertarClientes(Cliente nuevoCliente);

    ...
}
```

Tanto la clase DAO Cliente como su operador equivalente deben implementar esta interfaz:

```
public class ClienteDao implements Datos {

    public Cliente obtenerClientePorDni(String dni){
      // codigo que obtiene un cliente por dni usando objetos de base de
      // datos como Resulset
      ...
      }

    public void insertarClientes(Cliente nuevoCliente){
      // codigo que inserta un cliente usando objetos de base de
      // datos como Resulset
      ...
      }

    ...
}
```

```
public class ClienteOperador implements Datos {

    public Cliente obtenerClientePorDni(String dni){
      // codigo que obtiene un cliente por dni usando objetos Gestor
      ...
      }
```

```
    public void insertarClientes(Cliente nuevoCliente){
      // codigo que inserta un cliente usando objetos Gestor
      ...
      }

    ...
}
```

Solo resta buscar las clases (como ClienteLogica) donde se utilice el DAO de clientes y sustituirlo por la implementación del Operador cliente.

```
public class ClienteLogica {

      // replazar este
      // private Datos datos = new ClienteDao();
      // por este
      private Datos datos = new ClienteOperador();
      ...
}
```

Los siguientes apartados se van a centrar en las clases Operador y como usar distintos frameworks para introducir la capa de cache en la aplicación. Una vez se sabe como son los operadores y como se van a incluir en la aplicación el resto (el gestor y los almacenes) ya se implementa según el gusto del programador.

3.1.1 Aspectos

La programación orientada a aspectos ofrece la posibilidad de interceptar la llamada a un método del código de una aplicación y cambiar dicha llamada por cualquiera de las siguientes opciones:

- Realizar otra lógica en su lugar. (Remplazar)
- Ejecutar el código del método al que llama la aplicación pero que se ejecute otro método antes y/o después. (Incluir funcionalidad)

La filosofía AOP es perfecta para incluir una capa de cache en la aplicación que use siempre entidades (pojos) como resultado de los métodos de una clase (clases tipo DAO generalmente). A estos métodos se le aplicara las opciones anteriores (remplazar y/o incluir funcionalidad) para integrar la capa de cache en la aplicación sin afectar a la funcionalidad.

3.1.1.1 AspectJ

Existen muchos frameworks que ofrecen la posibilidad de usar programación orientada a aspectos y de entre todos ellos el más recomendable es AspectJ porque parece el más maduro, el que tiene más futuro y el que tiene mejor integración con Spring.

AspectJ tiene su propio lenguaje, muy parecido a Java, pero no es obligatorio utilizarlo gracias a un conjunto de Anotaciones Java que ofrece el framework de AspectJ que son capaces de transformar una clase Java en un “aspecto”, lo que evita tener que modificar el compilador de la IDE de Java que se esté utilizando para poder utilizar aspectos. El único inconveniente de esta opción

es que a la hora de generar los .class habrá que utilizar el compilador de AspectJ.
A continuación se describe como crear una clase Java que sea un “aspecto” y sirva para implementar una capa de cache para una aplicación como se ha descrito en el punto anterior.

Se crea una clase marcada como aspecto (con una anotación) que será el Operador de cache. AspectJ va a servir para crear las clases Operador y que el uso de una capa de cache no afecte a la funcionalidad de la aplicación. La implementación del gestor y de los almacenes es igual que en el punto anterior por lo que se va a omitir.

Lo más recomendable es implementar una clase (un aspecto) Operador por cada clase DAO y en cada Operador crear un método con la anotación “Pointcut” por cada método del DAO correspondiente que se quiere cachear. Por cada método marcado con PointCut se crea otro método marcado con la anotación “Around” para implementar la lógica de la cache y manejar el método del DAO sustituido ya que puede que sea necesario que se ejecute para no perder la funcionalidad.

Si el método que se quiere cachear es una consulta entonces su funcionalidad se puede sustituir completamente por la equivalencia en la cache pero si el método es una inserción, un borrado o una actualización se debe ejecutar la lógica de la cache pero también la lógica del método, por eso es importante tener siempre la posibilidad de poder ejecutar el método cacheado.

Se puede ver en el siguiente ejemplo. Se dispone de las siguientes clases DAO:

```
package ejemplo.dao;
public class ClienteDao {

    /*
     * Este metodo busca el registro de la tabla Cliente
     * que tiene el campo Id igual al parametro del metodo id. Si el
     * Cliente no existe devuelve nulo.
     * @param identificador del cliente
     * @return Objeto tipo Cliente abstracción al registro encontrado
     *         en base de datos
     */
    public Cliente buscarCliente (String id){
      ...
    }

     /*
     * Este metodo inserta un registro en la tabla Cliente de base de datos
     * Si el cliente ya existe eleva una excepcion
     * @param Objeto tipo Cliente abstracción al registro que se va
     *         a guardar en base de datos
     */
    public void insertarCliente (Cliente cliente) throws Exception{
      ...
   }
}
```

```
package ejemplo.dao;
public class FacturaDao {
    /*
     * La clase factura es la abstracción de un registro de la tabla
     * Factura. Este metodo busca todos los registros de la tabla factura
     * cuyos registros coinciden con los campos no nulos del parametro de
     * tipo Factura. Estos registros los devuelve en una lista de objetos
     * Factura. Si no hay facturas entonces devuelve nulo
     * @param Factura objeto cuyos campos equivalen a las columnas de la
     * tabla Factura y que se utilizaran como cirterio de la consulta
     * @return listas de objetos Factura que equivalen a los registros de
     * bd que coinciden con los criterios que vienen como parametro
     */
    public List<Factura> buscarFacturas (Factura factura){    ...
    }
    /*
     * Metodo que borra la factura cuyo id coincide con el parametro id
     * si la factura no existe entonces elava una excepción.
     * @param valor del identificador del registro de la tabla factura que
     * se quiere borrar
     */
    public void borrarFactura (Integer id) throws Exception{    ...
    }
}
```

```
@Aspect
public class OperadorCliente {

    private GestorCache gestorCache;

    @Pointcut("buscarCliente(* *.*(..))")
    private void busquedaCliente() {}

    @Around("busquedaCliente()")
    public Object buscarClientes(ProceedingJoinPoint proceedingJoinPoint){
        String id = (String)proceedingJoinPoint.getArgs()[0];

        Cliente cliente = gestorCache.buscarCliente(nuevoCliente.getId());

        if (cliente == null){
            return null;
        }else
            return cliente;
    }

    @Pointcut("insertarCliente(* *.*(..))")
    private void nuevosClientes() {}
      @Around("nuevosClientes()")
    public Object insertarNuevosClientes(ProceedingJoinPoint
proceedingJoinPoint){
        Cliente nuevoCliente = (Cliente)proceedingJoinPoint.getArgs()[0];

        Cliente cliente = gestorCache.buscarCliente(nuevoCliente.getId());
    if (cliente == null){
            gestorCache.insertarCliente(nuevoCliente);

            proceedingJoinPoint.proceed();
        }else
      throw new Exception ("el cliente ya existe");
    }
```

Métodos que podría tener el gestor para suportar la estrategia que se ha seguido. (Cache completa, numero de almacenes, etc..)

Ejecutar la función original para que se guarde el cliente en base de datos

Como se puede comprobar, la clase Operador es la responsable de asegurarse que la funcionalidad que ofrece la cache es la misma que ofrecen los métodos que han sido cacheados.

```
@Aspect
public class OperadorFactura {

    private GestorCache gestorCache;

    @Pointcut("buscarFacturas(* *.*(..))")
    private void busquedaFactura() {
    }

    @Around("busquedaFactura()")
    public Object buscarFacturas(ProceedingJoinPoint proceedingJoinPoint){
        Factura criterio = (Factura)proceedingJoinPoint.getArgs()[0];

        List<Factura> facturas = null;
        List<Factura> facturasAux = ArrayList<Factura>();

        if (criterio.getId() != null ){

            facturas = new ArrayList<Factura>();
            facturas.add(gestorCache.buscarFactura((criterio.getId()));
            return facturas;
        }
        // criterios que pueden devolver mas de una factura
        // criterio mas restrictivo
        if (criterio.getFecha() != null){
            facturas = gestorCache.buscarFacturaXFecha(criterio.getFecha());

        if (criterio.getTipo () != null){
            if (factures == null)
                  facturas = gestorCache.buscarFacturaXTipo(criterio.getTipo());
            for (Factura factura:facturas){
                if (factura.getTipo().equals(criterio.getTipo ()))
                    facturasAux.add(factura);
            }
            facturas.removeAll();
            facturas.addAll(facturasAux);
            facturasAux.removeAll();
        }

        if (criterio.getDestino () != null){

            if (factures == null)
            facturas = gestorCache.buscarFacturaXDestino(criterio.getDestino());

            for (Factura factura:facturas){
                if (factura.getDestino().equals(criterio.getDestino ()))
                    facturasAux.add(factura);
            }
            facturas.removeAll();
            facturas.addAll(facturasAux);
            facturasAux.removeAll();
        }
        return facturas;
    }

    @Pointcut("borrarFactura(* *.*(..))")
    private void borradoFacturas() {
    }
```

```
    @Around("borradoFacturas()")
  public Object borrarFactura(ProceedingJoinPoint proceedingJoinPoint){
        String id = (String)proceedingJoinPoint.getArgs()[0];

        Factura factura = gestorCache.buscarFactura(id);

        if (factura == null)
            throw new Exception ("el cliente ya existe");
        else{
                    gestorCache.borrarFactura(id);
            proceedingJoinPoint.proceed();
        }
    }
}
```

Como se puede comprobar el método “buscarFacturas” del operador ya es más complejo que el resto porque debe remplazar la funcionalidad que daría una consulta por criterios no nulos en base de datos. Otra posible manera de haber implementado la búsqueda para evitar recorrer la lista varias veces es la siguiente:

```
…
   @Around("busquedaFactura()")
    public Object buscarFacturas(ProceedingJoinPoint proceedingJoinPoint){
        Factura criterio = (Factura)proceedingJoinPoint.getArgs()[0];

        boolean cumpleFecha, cumpleTipo, cumpleDestino;
        List<Factura> facturas = gestorCache.buscarTodasFacturas();
        List<Factura> facturasAux = ArrayList<Factura>();

        for (Factura factura:facturas){

            if (criterio.getFecha() == null ||
                 criterio.getFecha().getTime()>factura.getFecha().getTime()   )
                cumpleFecha = true;
            else
                cumpleFecha = false;

            if (criterio.getTipo() == null ||
                 criterio.getTipo().equals(factura.getTipo())   )
                cumpleTipo = true;
            else
                cumpleTipo = false;

            if (criterio.getDestino() == null ||
                 criterio.getDestino().equals(factura.getDestino())   )
                cumpleDestino = true;
            else
                cumpleDestino = false;

            if (cumpleFecha && cumpleTipo && cumpleDestino)
                facturasAux.add(factura);
        }

        return facturasAux;
    }
…
```

Este método parece más eficiente desde el punto de vista programático pero si gracias al **conocimiento del negocio** se puede saber que la búsqueda solo con las fechas va a devolver un objeto "List" muy pequeño y de forma rápida (por ejemplo, debido al **conocimiento del negocio** es conocido que el grupo de facturas por fecha es muy usado y se dispone de un Almacen para este concepto en el Gestor) entonces será mejor hacer primero la búsqueda por fechas y luego hacer varias consultas pequeñas ya que, por lo general, ***es mejor hacer varias búsquedas en listas de POJOS pequeñas que hacer una en una lista de POJOS muy grande***.

El conocimiento del negocio es esencial para implementar un sistema de cache eficiente ya que permite definir una estrategia acertada frente a la que parece más correcta programáticamente. No hay que olvidar NUNCA que lo que hace eficiente a un sistema de cache es:

- Disponer de la información que requiere la aplicación en una zona de rápido acceso (premisa tradicional).
- **Acceder a la información disponible de una manera eficiente.**

3.1.1.2 Spring

Si se está utilizando Spring en la aplicación donde se quiere implementar el sistema de cache, o no importa incluirlo, entonces se debe saber que Spring utiliza AOP y permite el uso de interceptores de llamadas a métodos que permitirían implementar el sistema de cache exactamente igual que en el apartado anterior con más ventajas como no tener que cambiar de compilador o que no hace falta aprender nada sobe filosofía AOP, en realidad se puede estar usando interceptores sin saber que se está usando AOP.

Para ilustrar como se utilizan los interceptores de Spring se va resolver el ejemplo del apartado anterior partiendo de las mismas clases DAO. Se va a partir del siguiente fichero de configuración de Spring:

```
<beans>
    <bean id="accionesCliente" class="ejemplo.logica.ClienteAccion">
        <property name="dao" value="daoCliente">
    </bean>

    <bean id="daoCliente" class="ejemplo.dao.ClienteDao"/>

    <bean id="accionesFactura" class="ejemplo.logica.FacturaAccion">
        <property name="dao" value="daoFactura">
    </bean>

    <bean id="daoFactura" class="ejemplo.dao.FacturaDao"/>
</beans>
```

Como se puede observar, se dispone de dos Bean (accionesCliente y accionesFactura) que centralizan todas las acciones sobre clientes y facturas y dos Bean DAO (daoCliente y daoFactura). Se debe crear dos interceptores y dos Bean Proxys, uno por cada Bean DAO, y sustituir la referencia a los Bean DAO por referencias a los nuevos Bean proxy.

```
<beans>
    <bean id="accionesCliente" class="ejemplo.logica.ClienteAccion">
        <property name="dao" value="proxyCliente">
    </bean>

    <bean id="daoCliente" class="ejemplo.dao.ClienteDao"/>

    <bean id="accionesFactura" class="ejemplo.logica.FacturaAccion">
        <property name="dao" value="proxyFactura">
    </bean>

    <bean id="daoFactura" class="ejemplo.dao.FacturaDao"/>

    <bean id="clienteInterceptor" class="ejemplo.cache.OperadorCliente"/>

    <bean id="facturaInterceptor" class="ejemplo.cache.OperadorFactura"/>
```

Parte Tradicional

Definición de interceptores

```
<bean id="clienteAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
      <property name="advice">
            <ref local="clienteInterceptor" />
         </property>
         <property name="patterns">
                <list>
                  <value>.*buscarCliente</value><value>.*insertarCliente</value>
                </list>
         </property>
   </bean>
<bean id="facturaAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
      <property name="advice">
            <ref local="facturaInterceptor" />
         </property>
         <property name="patterns">
                <list>
                  <value>.*buscarFacturas</value><value>.*borrarFactura</value>
                </list>
         </property>
   </bean>
<bean id="proxyCliente"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
        <value>daoCliente</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>clienteAdvisor</value>
      </list>
    </property>
  </bean>
<bean id="proxyFactura"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
        <value>daoFactura</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>facturaAdvisor</value>
      </list>
    </property>
  </bean>
</beans>
```

```
public class OperadorCliente implements MethodInterceptor {

    private GestorCache gestorCache;

    public Object invoke(MethodInvocation methodInvocation) throws Throwable{

        if (methodInvocation.getMethod().getName().equals("buscarCliente")){
        ...
        }

        if (methodInvocation.getMethod().getName().equals("insertarCliente")){
        ...
        }
    }
}
```

Este código sería igual que en el apartado anterior

```
public class OperadorFactura implements MethodInterceptor {

    private GestorCache gestorCache;

    public Object invoke(MethodInvocation methodInvocation) throws Throwable{

        if (methodInvocation.getMethod().getName().equals("buscarFacturas")){
        ...
        }

        if (methodInvocation.getMethod().getName().equals("borrarfactura")){
        ...
        }
    }
}
```

Este código sería igual que en el apartado anterior

3.2 EHCACHE

La versión de EHCACHE que se ha utilizado es la 2.3.2 que es la versión más reciente que existía cuando se escribió este manual.

El punto de partida de una cache EHCACHE es el fichero de configuración ehcache.xml donde se le indica uno a uno los almacenes de datos que se van a utilizar. Cada almacén de datos es un Map donde por medio de un índice (key) se accede a un elemento (Element) donde se guardar el objeto que se ha guardado en el almacen. Cada almacén es independiente de otro almacén y se le pueden definir propiedades como el tiempo que duran en memoria, el número de elementos en memoria, si ha de guardarse en disco o si hay el objeto que se devuelve es una copia o el objeto que esta cacheado.

Ejemplo:

```
<cache name="clienteCache" maxElementsInMemory="100"
eternal="true" timeToIdleSeconds="10" timeToLiveSeconds="50"
overflowToDisk="false"copyOnRead="true"
copyOnWrite="false">

<copyStrategy class="prueba.ClienteCopia"/>
<cacheWriter>
<cacheWriterFactory class="prueba.ClienteEscribe"/>
</cacheWriter>
</cache>
```

Simple, Verdad? Ahora ya se puede instanciar el CACHE en código y guardar objetos.

```
…
Cliente cliente = new Cliente();
…
CacheManager manager = new CacheManager();
Cache cache = manager.getCache("clienteCache");
Element element = new Element(cliente.getId(),
cliente);
cache.put(element);
…
```

Un objeto “Element” puede ser tanto una entidad (un pojo de la aplicación) como una agrupación de entidades. Es importante denotar esta diferencia porque las estrategias de cacheo de las implementaciones actuales van orientadas a cachear una entidad (un pojo, dto, por ejemplo) o el resultado de una función (cachear una query por ejemplo). En el ejemplo anterior, la clave para cachear la entidad era un campo único (`Element element = new Element(cliente.getId(),cliente);`) para almacenar una función (como se va a ver en el siguiente apartado) se suele utilizar una combinación del nombre de la función y los parámetros que se le pasan a la función ya que es lógico pensar que una función siempre devolverá los mismo si se le invoca con los mismos parámetros.

La propiedad copyOnRead permite indicar que al leer un determinado elemento de la cache se va a llamar a una clase CopyStrategy (ReadWriteCopyStrategy para la próximas versiones) para reconstruir el objeto que va a devolver la cache. Esta propiedad permite modificar el objeto que va a devolver la cache y permite generar una copia y devolverla a la aplicación para que si es modificada por la aplicación esta modificación no se refleje en memoria (y asi evitar el efecto comentado al final del punto 2).

El elemento cacheWriter permite definir cómo se van escribir y a borrar los elementos en cache. Todas las acciones pasaran por la clase creada por la factoría cacheWriterFactory. La manera más rápida de crear una clase que sirve para controlar la escritura/lectura de objetos en cache es la siguiente:

```
public class ClienteEscribe extends CacheWriterFactory
{
    public CacheWriter createCacheWriter(Ehcache arg0, Properties arg1) {
        return new CacheWriter() {

            public CacheWriter clone(Ehcache arg0) throws
            CloneNotSupportedException {
            }

            public void init() {
            }

            public void dispose() throws CacheException {
            }
```

```
            public void write(Element arg0) throws CacheException {
            // este metodo es el mas importante. Con el se puede
            // modificar no solo el objeto que se va a guardar sino
            // tambien la clave. No hay que olvidar que es la clave la
            // que indica si el objeto esta en cache o hay que hacer
            // una consulta para recuperarlo.
            }
            public void writeAll(Collection<Element> arg0) throws CacheException {
            }

            public void delete(CacheEntry arg0) throws CacheException {
            }

            public void deleteAll(Collection<CacheEntry> arg0) throws
            CacheException {
            }
        };
    }
}
```

3.2.1 SPRING

EHCACHE dispone de sus propias etiquetas de Spring para configurar los accesos a cache desde el código sin tener que modificar el código original.
Por ejemplo, si se dispone del siguiente fichero de configuración de Spring:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean class="prueba.negocio.ClienteOperaciones"
name="beanClienteNegocio" >
        <property name="clienteDao" ref="beanClienteDao">
</bean>
<bean class="prueba.dao.ClienteDao" name="beanClienteDao" />
</beans>
```

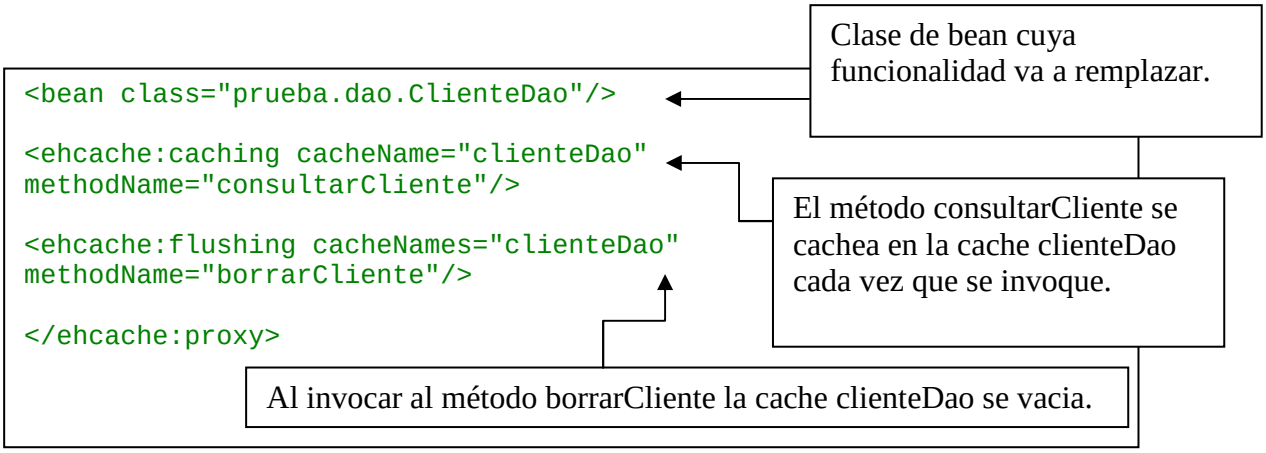
Se debe incluir al fichero de configuración los siguientes namespaces y las siguientes etiquetas:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ehcache="http://www.springmodules.org/schema/ehcache"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springmodules.org/schema/ehcache
http://www.springmodules.org/schema/cache/springmodules-ehcache.xsd">
```

Fichero de configuración donde se definen los almacenes de datos CACHE de EHCACHE.

```
<ehcache:config configLocation="classpath:ehcache.xml" />

<ehcache:proxy id="clienteCacheProxy">
```

Donde se invoque el bean ClienteDao hay que cambiarlo por este proxy.



Una vez introducidas las nuevas etiquetas, se elimina el Bean beanClienteDao y se debe utilizar el Bean clienteCacheProxy como si fuese el beanClienteDao.

```
...
<bean class="prueba.negocio.ClienteOperaciones"
name="beanClienteNegocio" >
        <property name="clienteDao" ref="clienteCacheProxy">
</bean>
...
```

El Bean clienteCacheProxy se comportará exactamente igual que el Bean beanClienteDao solo que cuando se llame al método consultar cliente dos veces con los mismo parámetros entonces la segunda vez no accederá a la base de datos si no que devolverá lo que tiene almacenado en cache.

Usando Spring también es más sencillo configurar el método por el cual se generan las claves con las que los objetos se guardan en la cache. Para definir como se generan las claves hay que incluir las siguientes líneas (en rojo) en el fichero de configuración de Spring:

```
<beans>

<ehcache:config configLocation="classpath:ehcache.xml" />

<bean name="generador"
class="org.springmodules.cache.key.HashCodeCacheKeyGenerator"/>

<ehcache:proxy id="clienteCacheProxy">

<ehcache:cacheKeyGenerator refId="generador"/>

<bean class="prueba.dao.ClienteDao"/>

<ehcache:caching cacheName="clienteDao"
methodName="consultarCliente"/>
....
```

Se puede utilizar implementaciones de Spring como la clase HashCodeCacheKeyGenerator que utiliza el método hashCode() del objeto que se va a guardar en cache para generar su clave o se podría utilizar cualquier clase de implementación propia que implemente la interfaz CacheKeyGenerator.

3.3 JPA. Hibernate. Cache de segundo Nivel

La versión de HIBERNATE que se ha utilizado es la 3.6.1 que es la versión más reciente que existía cuando se escribió este manual.

JPA ofrece la posibilidad de usar cache de segundo nivel que es una cache de datos como la que se está describiendo en este manual.
Para ver como se usa, se va a describir un ejemplo donde se va a utilizar la implementación de JPA de Hibernate y para la implementación del sistema de cache se utilizará EHCACHE (descrita en el apartado anterior).
Para un sistema con la entidad Cliente:

```
@Entity
@Table(name = "cliente")
@NamedQueries({@NamedQuery(name = "Cliente.findById", query = "SELECT c
FROM Cliente c WHERE c.id = :id)
public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "id", nullable = false)
    private Integer id;
    @Column(name = "nombre")
    private String nombre;

    public Cliente() {
    }

    public Cliente(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

Lo primero es configurar JPA con Hibernate, para ello se debe colocar en el classpath un fichero persistence.xml como el siguiente:

```
<persistence  >
<persistence-unit name="sample"  transaction-type="RESOURCE_LOCAL">

      <class>prueba.dominio.Cliente</class>
```

```
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.diale
      ct.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
       </properties>
</persistence-unit>
</persistence>
```

Se a aplicara un ejemplo. Se dispone del DAO ClienteDao donde el método buscarClientePorId busca una entidad y el método buscarClientePorId2 realiza una consulta:

```
public class Dao {

    private static EntityManager entityManager;

    public Dao(){

        if (entityManager == null){
            EntityManagerFactory emf =
Persistence.createEntityManagerFactory("sample");
            entityManager = emf.createEntityManager();
        }
    }

    public EntityManager getEntityManager() {
        return entityManager;
    }
}
```

```
public class ClienteDao extends Dao{

   public Cliente buscarClientePorId (Integer id){
      // esta sentencia es importante ya que si no se vacia el
      // contenedor entityManager a las pruebas posteriores le puede
      // afectar la cache de primer nivel
      getEntityManager().clear();

      Cliente cliente = getEntityManager().find(Cliente.class, id);


      return cliente;
   }

   public Cliente buscarClientePorId2 (Integer id){

      // esta sentencia es importante ya que si no se vacia el
      // contenedor entityManager a las pruebas posteriores le puede
      // afectar la cache de primer nivel
      getEntityManager().clear();

       Query query =
getEntityManager().createNamedQuery("Cliente.findById");
       query.setParameter("id", id);
       Cliente cliente =  (Cliente)query.getSingleResult();

       return cliente;
   }
}
```

Si se realiza la siguiente prueba:

```
ClienteDao clienteDao = new ClienteDao();

Cliente resultado =  clienteDao.buscarClientePorId2(1);

System.out.println ("el resultado es " + resultado.getNombre());

resultado =  clienteDao.buscarClientePorId2(1);

System.out.println ("el resultado es " + resultado.getNombre());
```

El resultado que se ve en la salida del sistema es:

```
Hibernate: select cliente0_.id as id1_, cliente0_.nombre as nombre1_
from cliente cliente0_ where cliente0_.id=? limit ?
el resultado es Cliente 1
Hibernate: select cliente0_.id as id1_, cliente0_.nombre as nombre1_
from cliente cliente0_ where cliente0_.id=? limit ?
el resultado es Cliente 1
```

Por lo tanto la consulta no está cacheada y se han realizado dos accesos a base de datos. Si se prueba con la búsqueda del elemento se obtendrá el mismo resultado.

```
ClienteDao clienteDao = new ClienteDao();

Cliente resultado =  clienteDao.buscarClientePorId(1);

System.out.println ("el resultado es " + resultado.getNombre());

resultado =  clienteDao.buscarClientePorId(1);

System.out.println ("el resultado es " + resultado.getNombre());
```

Para activar la cache de segundo nivel de JPA se debe indicar en el fichero persistence.xml las siguientes propiedades (en color rojo) para indicar la implementación que se va a utilizar:

```
…
<properties>

  <property name="hibernate.dialect" value="org.hibernate.dialect.HSQ
LDialect"/>
<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
<property name="hibernate.cache.provider_class"
value="net.sf.ehcache.hibernate.SingletonEhCacheProvider"/>
<property name="hibernate.cache.use_query_cache" value="true"/>
<property name="hibernate.cache.use_second_level_cache"
value="true"/>
<property name="net.sf.ehcache.configurationResourceName"
value="ehcache.xml" />
```

Hibernate configurara el sistema de cache como se ha visto en el apartado anterior por lo que el fichero ehcache.xml debe estar en el classpath. Ahora solo falta indicar que la entidad es cacheable y en que cache se quiere guardar para que el método find del Entity Manager la encuentre (líneas en rojo):

```
@Cache(usage =
CacheConcurrencyStrategy.NONSTRICT_READ_WRITE,region="clienteCache")
@Entity
@Table(name = "cliente")
@NamedQueries({@NamedQuery(name = "Cliente.findById", query = "SELECT
c FROM Cliente c WHERE c.id = :id"), @NamedQuery(name =
"Cliente.findByNombre", query = "SELECT c FROM Cliente c WHERE
c.nombre = :nombre")})
public class Cliente implements Serializable {
….
```

Si lo que se desea es cachear la consulta lo que se debe hacer en indicar que es cacheable al ejecutarla (líneas en rojo):

```
…..
public Cliente buscarClientePorId2 (Integer id){
       getEntityManager().clear();
       Query query =
getEntityManager().createNamedQuery("Cliente.findById");
       query.setParameter("id", id);
       Cliente cliente =
(Cliente)query.setHint("org.hibernate.cacheable",
true).getSingleResult();

       return cliente;
   }
….
```

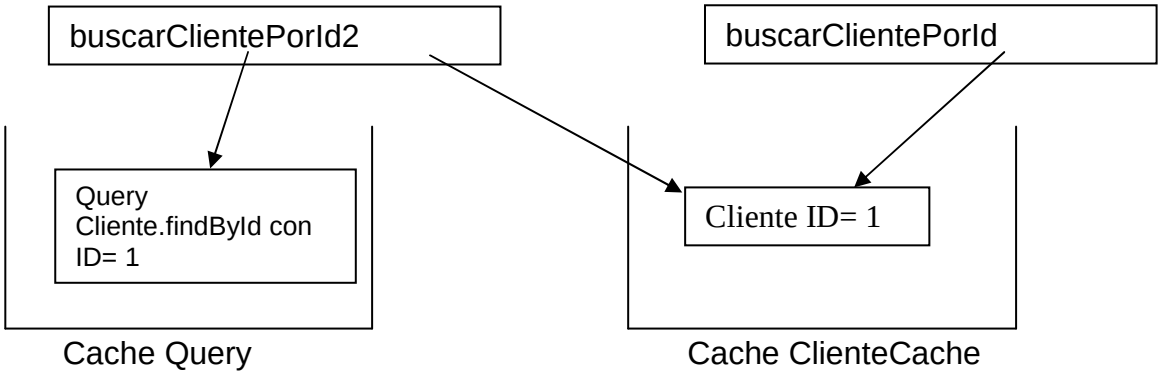
Ahora si volvemos a realizar la prueba, tanto con buscarClientePorId como con buscarClientePorId2 el resultado va a ser:

```
Hibernate: select cliente0_.id as id1_, cliente0_.nombre as nombre1_
from cliente cliente0_ where cliente0_.id=? limit ?
el resultado es Cliente 1
el resultado es Cliente 1
```

¿Qué es lo que ha pasado? Cuando Hibernate ha cargado Ehcache ha creado tres caches:

- ClienteCache:Tal y como se le ha indicado en el fichero de configuración ehcache.xml
- Query Cache: Una cache para las consultas.
- Update Cache: Una cache para mejorar las actualizaciones.

Cuando se ejecuta buscarClientePorId2 se crea un registro en la cache “clienteCache” y otro en la cache “query” y cuando se ejecuta buscarClientePorId solo se crea un registro en la cache “clienteCache”.



Este comportamiento tiene un efecto secundario que merece la pena denotar y es que si se ejecuta primero buscarClientePorId2 al ejecutar de nuevo buscarClientePorId2 o buscarClientePorId se obtendrá el registro de cache y no se accederá a base de datos.

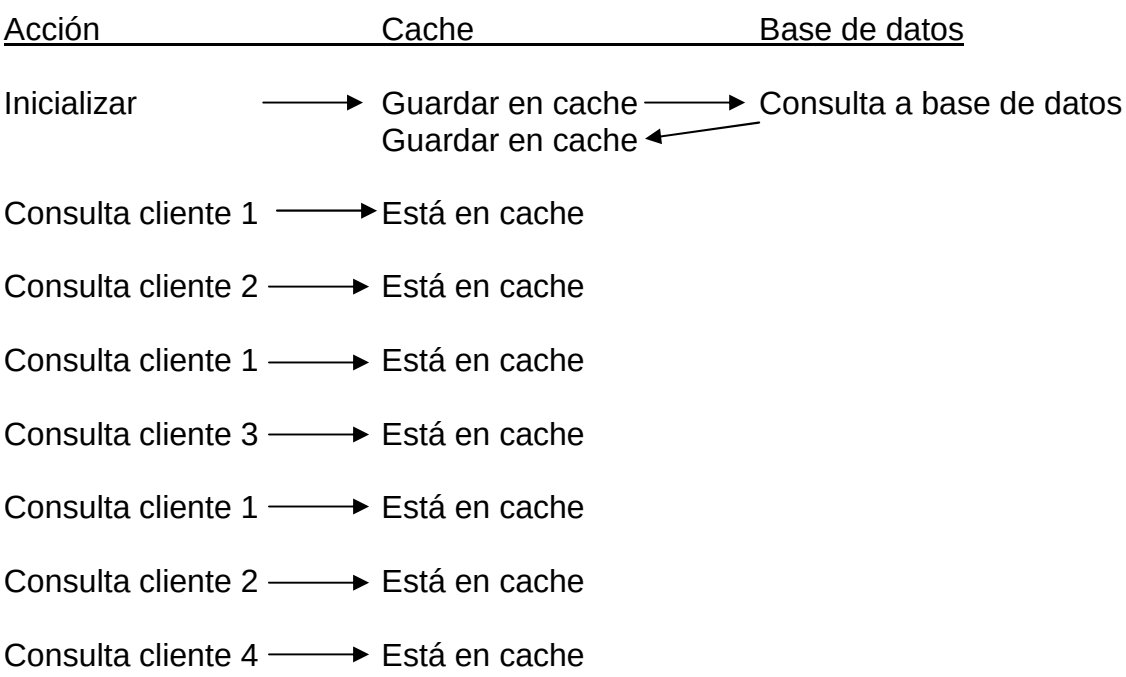
Cabe también destacar que se puede acceder a la cache directamente con la sentencia:

```
CacheManager cache = CacheManager.getInstance();
```

Si se accede directamente a cache se pueden realizar acciones como borrar entradas en cache que ya no se usan, actualizar entradas que se han quedado desfasadas, etc.. Pero la más recomendable es la de inicializar entradas en la cache. Esta operación se basa en el hecho intangible de que “es mejor una consulta algo pesada a base de datos que 10000 consultas ligeras”, mejor verlo en un ejemplo. En la tabla Cliente hay 5 elementos y se sabe que el sistema tiene suficiente memoria para almacenar todos los clientes. No se inicializan entradas y se realiza la siguiente ejecución de órdenes:

Acción	Cache	Base de datos
Consulta cliente 1 →	No está en cache →	Consulta a base de datos
	Guardar en cache ←	
Consulta cliente 2 →	No está en cache →	Consulta a base de datos
	Guardar en cache ←	
Consulta cliente 1 →	Esta en cache	
Consulta cliente 3 →	No está en cache →	Consulta a base de datos
	Guardar en cache ←	
Consulta cliente 1 →	Esta en cache	
Consulta cliente 2 →	Esta en cache	
Consulta cliente 4 →	No está en cache →	Consulta a base de datos
	Guardar en cache ←	

Se han realizado 4 consultas a base de datos mientras que si se hubiera hecho una consulta inicial y se hubiesen inicializado las cuatro entradas en cache el resultado final seria de solo una consulta:



4. Planteamiento

Siendo realistas, en el desarrollo de una aplicación Java solo se plantea el uso de cache al final del desarrollo, ese es el motivo por el que su uso es poco frecuente. Por eso el uso de los frameworks y el acercamiento que sugiere este manual (uso de aspectos) están pensados para “incrustar” un sistema de cache posterior a que toda la aplicación este diseñada y desarrollada (o al menos una parte muy significativa). En este apartado se va describir planteamientos de cómo usar cache en casos muy generales siguiendo esta filosofía pero también se va a describir planteamientos que son favorables para el cacheo de la información (ojo, siempre compatibles con una aproximación al problema de forma tradicional).

4.1 No Tradicional

Sea cual sea la aplicación que se quiera desarrollar, una pregunta que es primordial, y que se ha de plantear al principio del desarrollo, es la siguiente:

¿Cuál de las siguientes definiciones se ajusta más a la aplicación que se quiere desarrollar?

- La aplicación es un software que envuelve un modelo de datos y en función de cómo este ese modelo de datos se irán desarrollando capas

de software hasta conseguir ofrecer al usuario lo que solicitó con esa aplicación.

- Ó, la aplicación es un software que mediante una interfaz recoge y ofrece la funcionalidad que el usuario requiere mediante un funcionamiento interno desarrollado por capas de software. Una de estas capas es donde se guarda y se recupera la información.

La segunda opción sin duda.

Es cierto que pueden haber casos en los que la base de datos venga impuesta. En estos casos tampoco hay mucha maniobra y si tenemos un elemento fijo desde el principio es mejor tenerlo en cuenta desde el principio.

La experiencia enseña que en la mayoría de proyectos la base de datos empieza con la aplicación, es mas suele ser previa a la implementación del código; Por lo que ¿No es un poco raro que uno de los primeros pasos al desarrollar una aplicación sea diseñar el modelo de datos?

Además, ese modelo de base de datos debe de ser muy limpio y fácil de entender.

Ejemplo, Si la aplicación es de facturación de clientes tenemos una tabla Cliente donde se guardan los datos del cliente y otra tabla Factura donde se guardan los datos de las facturas del cliente y una calve ajena a la tabla Cliente.

La base de datos no es el núcleo central de una aplicación sino que es la parte de la aplicación donde se guardan los datos. Si para que la aplicación funcione mejor se necesita que el modelo de datos no sea limpio ni fácil de entender se debe cambiar el modelo de datos. Esta filosofía se sigue en aplicaciones tipo “Gestor documental”, en estas aplicaciones las tablas son para índices de búsqueda, para guardar ficheros, etc..y si son muy difíciles de interpretar sin ver el código.

Ejemplo, si tengo una aplicación web con una pantalla de alta de cliente entonces lo importante es que si el usuario quiere dar de alta un cliente la información se guarde. Que mas dará que se guarde en una tabla Cliente o en tres tablas Cliente_Nombre, Cliente_Indice y Cliente_Activo sin con eso consigo que la aplicación vaya más rápido, o sea más fiable, o sea más escalable, etc…

Es importante no hacer al desarrollado esclavo del modelo de datos que sigue la base de datos; se debe desarrollar la aplicación para alcanzar el objetivo deseado y ya habrá tiempo para ver como se guarda y recupera la información.

Una vez que ya se tienen claras las tablas, es decir se ha definido el modelo de datos como mejor se puede usar por la aplicación, entonces se deben analizar las características de las tablas (frecuencia y modo de uso, tamaño, etc..) y clasificar las tablas como “Tipo” y “De negocio”. Una vez clasificadas su modo

de guardar en cache es igual que las tablas “Tipo tradicional” y “De negocio tradicional” que se describen a continuación

4.2 Tablas Tipo Tradicional

Por sus características, una tabla Tipo siempre se debe cachear de forma completa (todos los datos en memoria). Esta tablas son las típicas tablas, sea cual sea la aplicación que se esté desarrollando, como “Provincias”, “Pais”, “Tipo de Documento”, “Tipo de Nivel”, etc.. Estas tablas no van a crecer, ni se van a modificar, mucho más que los datos iniciales con los que se han cargado. En la ya conocida aplicación de ejemplo de este manual “Facturas de Cliente” se le va a añadir una nueva entidad “TipoCliente” que representa los diferentes tipos de cliente existente.

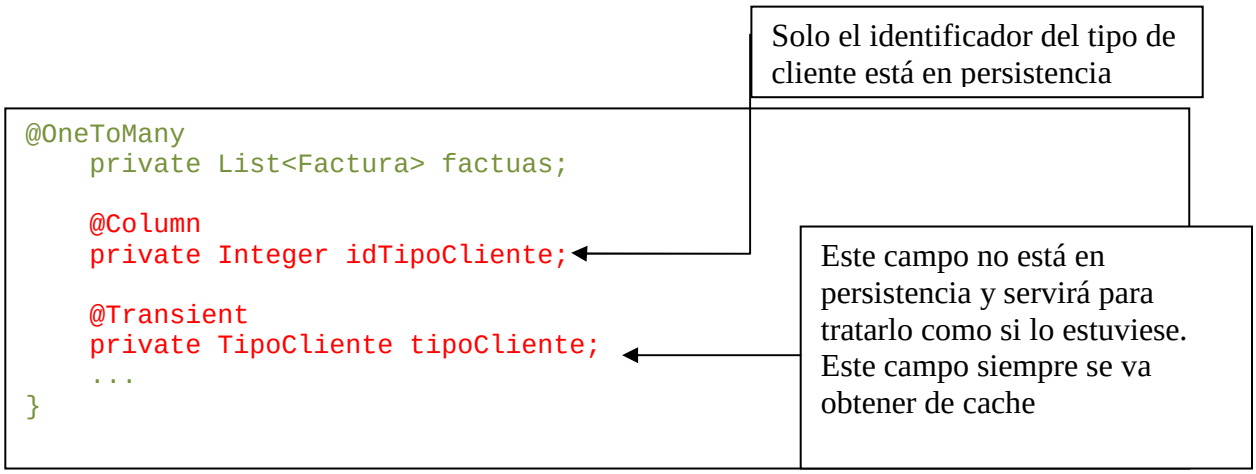
Se va a suponer que se está utilizando JPA ya que así queda ilustrado el caso de aplicaciones que usan JPA y de aplicaciones que no usan ORM y utilizan métodos tipo “fillClient” para rellenar los POJOs.

Se crea la entidad TipoCliente:

```
@Entity
public class TipoCliente {

    @Id
    @Column
    private Integer id;
    @Cloumn
    private String descripcion;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
}
```

Se incluyen dos campos (en color rojo) en la entidad Cliente:

```
@Entity
public class Cliente {

    @Id
    @Column
    private Integer id;

    @Column
    private String Nombre;
```



Ahora se debe crear un DAO para el tipo de Cliente. Este DAO debe de tener también una clase operador de cache ya que una vez inicializadas todas las entradas en cache (todos los tipo cliente estarán en cache) ya no se accederá a base de datos.

```
public class TipoClienteDao {

    @PersistenceContext
    EntityManager entityManager;

    public List<TipoCliente> buscarTodosTiposCliente(){
        Query query = entityManager.createQuery("SELECT e FROM
TipoCliente e");
        return (List<TipoCliente>) query.getResultList();
    }

    public TipoCliente buscarTipoCliente(Integer id){
        Query query = entityManager.createQuery("SELECT e FROM
TipoCliente e where e.id = :id");
        query.setParameter ("id",id);
        return (TipoCliente) query.getSingleResult();
    }


    public void insertarTipoCliente(TipoCliente tipoCliente){
        entityManager.persist(tipoCliente);
        entityManager.flush();
    }
}
```

```
public class OperadorTipoCliente implements org.aopalliance.intercept.MethodInterceptor{

      private GestorCache gestorCache;

      private boolean yaInicializado;

      private void inicializar(MethodInvocation methodInvocation) throws Throwable {

      List<TipoCliente> salida = (List<TipoCliente>)methodInvocation.proceed();

      if (salida != null)
            for (TipoCliente tipoCliente: (List<TipoCliente>)salida)
                  gestorCache.insertarTiposCliente (tipoCliente);

}
```

```
      public Object invoke(MethodInvocation methodInvocation) throws Throwable {
       Object salida = null;

      if (!yaInicializado){
            yaInicializado = true;
            inicializar(methodInvocation);
      }

      if (methodInvocation.getMethod().getName().equals("buscarTipoCliente")){

            List resultado =
gestorCache. encontrarTiposCliente ((Integer)methodInvocation.getArguments()[0]);

            if (resultado != null)
                  salida = resultado.get(0);
      }

      if (methodInvocation.getMethod().getName().equals("buscarTodosTiposCliente")){

            salida = gestorCache.encontrarTiposCliente(null);

      }
              if (methodInvocation.getMethod().getName().equals("insertarTipoCliente")){
gestorCache.insertarTiposCliente((TipoCliente)methodInvocation.getArguments()[0]);
            methodInvocation.proceed();
        }

        return salida;
    }

}
```

Lo más recomendable es poner el mismo nombre que se usa en las funciones del DAO a los métodos del gestor de cache siempre y cuando hagan lo mismo (como es el caso de de “insertarTipoCliente”). Si la función es parecida pero no es exactamente la misma el nombre de la función será parecido pero NO debe de ser el mismo (como es el caso de las funciones “buscar???” del DAO y la función “encontrarTiposCliente” ya que esta última hace lo mismo que las dos anteriores juntas).

```
public class GestorCache {

    private AlmacenTipoCliente almacenTipoCliente;

    public List encontrarTiposCliente(Integer id) {
        List salida ;
        if (id == null){
            Collection todos = almacenTipoCliente.cogerTodos();
            if (todos != null)
                salida = new ArrayList(todos);
            else
                salida = null;
        }else{
```

```
            TipoCliente tipoCliente = almacenTipoCliente.coger(id);
            if (tipoCliente != null){
                salida = new ArrayList();
                salida.add(tipoCliente);
            }else
                salida = null;
        }
        return salida;
    }

  public void insertarTiposCliente(TipoCliente tipoCliente) {
       almacenTipoCliente.insertar(tipoCliente);
    }

…… // resto de funciones para otros operadores
}
```

Se ha omitido la parte de control de concurrencia para simplificar el código pero en una implementación real hay que incluirla como se indico en el apartado 3.1.

```
public class AlmacenTipoCliente {

    private HashMap<Integer, TipoCliente> indicePorId;

    public  AlmacenTipoCliente() {
        indicePorId = new HashMap<Integer, TipoCliente>();
    }
      public TipoCliente coger(Integer id) {
        return indicePorId.get(id);
    }

    public Collection cogerTodos() {
        return indicePorId.values();
    }

    public void insertar(TipoCliente tipoCliente) {
        indicePorId.put(tipoCliente.getId(), tipoCliente);
    }
}
```

A nivel de código solo resta incluir la llamada a la función del DAO TipoCliente correspondiente cada vez que se necesite el tipo de cliente. Como por ejemplo:

```
public class ClienteDao extends Dao{

   private TipoClienteDao tipoClienteDao;

   public Cliente buscarClientePorId (Integer id){
      getEntityManager().clear();

      Cliente cliente = getEntityManager().find(Cliente.class, id);

      Cliente.setTipoCliente(tipoClienteDao.
buscarTipoCliente(cliente.getIdTipoCliente()));

      return cliente;
   }
   …
}
```

A nivel de la aplicación en general, en este punto las clases Gestor, Operador y Almacen no se utilizan para nada y el código añadido (en rojo) en la clase CielnteDAO provocaría una llamada a base de datos para obtener el tipo de cliente. En importante que la capa de cache este completamente separada del resto de la aplicación ya que no hay que olvidar que:

- La cache se debe poder DESACTVAR EN CUALQUIER MOMENTO y la aplicación seguiría funcionando igual.
- Una separación tan clara hace fácil discernir si ante un fallo de funcionamiento (por ejemplo, borrar un cliente y al buscarlo que siga apareciendo) el fallo es de la aplicación o está en un manejo erróneo del sistema de cache.

Ahora solo resta definir los interceptores de Spring y el sistema de cache implementado ya estaría funcionando.

```
<beans>
    <bean id="daoCliente" class="ejemplo.dao.ClienteDao">
        <property name="tipoClienteDao" value="proxyTipoCliente">
    </bean>

    <bean id="daoTipoCliente" class="ejemplo.dao.ClienteDao"/>

<bean id="tipoClienteInterceptor" class="ejemplo.cache.OperadorTipoCliente"/>
```

```
<bean id="tipoClienteAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
      <property name="advice">
            <ref local="tipoClienteInterceptor" />
         </property>
         <property name="patterns">
                <list>
                  <value>.*buscarTipoCliente</value>
                 <value>.*buscarTodosTiposCliente</value>
                  <value>.*insertarTipoCliente</value>
                </list>
         </property>
   </bean>
<bean id="proxyTipoCliente"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
        <value>daoTipoCliente</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>tipoClienteAdvisor</value>
      </list>
    </property>
  </bean>
</beans>
```

4.3 Tablas De Negocio Tradicional

Estas tablas tienen el inconveniente de que crecen hasta el infinito :(. Al crecer de forma indefinida no se pueden cachear porque por lo general no se dispone

de tanta memoria de acceso rápida (RAM del equipo, memoria distribuida, etc..) como de memoria dispone una base de datos. Para resolver este problema se debe buscar una aproximación en la que solo se cachea la información que le hace falta a la aplicación para que funcione más rápido y no toda la información posible.

En el punto anterior se ha podido ver que clases hacen falta para cachear información. Este punto se va a centrar en como plantear un problema con Tablas de negocio para poder cachear la tabla de forma completa.

Para encontrar una aproximación que realmente mejore el rendimiento es necesario conocer el uso que los usuarios dan a la aplicación. Es decir, se necesita ver como el usuario utiliza la aplicación, que información suele utilizar más, que búsquedas son las más frecuentes, etc..

Una vez que se conoce el uso de la aplicación se debe buscar los siguientes casos en la aplicación:

4.3.1 Caso óptimo

Existe un conjunto de datos más utilizado que otro en una tabla de negocio y este conjunto de datos es cache completa. Este caso es un claro ejemplo de lo que se indica en el punto 4.1 y se ha hecho una división de la información atendiendo más a que sean comprensibles las tablas de base de datos que a la funcionalidad de la aplicación.

Se dispone de la tabla "Cliente" que no es cacheable completa. Esta tabla dispone de un campo que es "Activado"; dicho campo indica si un cliente está activo o no. El número de clientes activos nunca supera los n elementos que si son un número de elementos que se pueden cachear y suponen la mayoría de accesos a la tabla.

En este caso se puede tratar exactamente como si fuese una tabla Tipo con la salvedad de que solo que se almacenarán los clientes activos. Se pueden realizar todas las operaciones en cache:

- Búsqueda unitaria. Se crea un índice en la clase Almacen y una función que busque por ese índice.

```
public class AlmacenCliente {

prívate Map<Integer,Cliente> clientesPorId;
…
public Cliente buscarPorId (Integer id){
        return clientesPorId.get(id);
    }
…
}
```

- Búsqueda de conjuntos. Se crea una función que recorra los clientes de cualquier índice y devuelva el conjunto que se solicita.

```
public class AlmacenCliente {

prívate Map<Integer,Cliente> clientesPorId;
…
public List<Cliente> buscarPorFecha (Date date){
      List<Cliente> salida = new ArrayList<Cliente> ();

      for (Cliente cliente: clientesPorId.values())
            if (cliente.getFecha().getTime() > date.getTime())
                  salida.add(cliente);

        return salida;
    }
…
```

- Inserción/Borrado/Modificación (reflejando la operación en base de datos).

Si alguna funcionalidad requiere del uso de clientes no activos entonces habrá que acceder a base de datos pero sin alterar la información en cache.

4.3.2 Caso aceptable

No existe una diferenciación tan clara como el caso anterior pero campos de la tabla pueden dar conjuntos muy utilizados. Este caso no se produce por un mal diseño del almacenamiento de datos si no por la tendencia de uso de los usuarios de la aplicación.

Se dispone de la tabla "Cliente" que no es cacheable completa. Esta tabla dispone de un campo que es "Provincia"; dicho campo indica la provincia del Cliente. El número de clientes de Madrid y Barcelona nunca superan los n elementos que sí son un número de elementos que se pueden cachear y suponen la mayoría de accesos a la tabla.

La gran diferencia con el caso anterior es que en el anterior el campo discriminatorio permite diferenciar claramente los conjuntos. En este no se puede por lo que se permiten las siguientes operaciones en cache:

- Búsqueda unitaria. Igual que en el caso anterior pero si no se encuentra un registro habrá que buscarlo en base de datos.

```
public class OperadorCliente {

prívate GestorCache gestorCache;

public Object invoke(MethodInvocation methodInvocation) throws
Throwable {
       Object salida = null;
      if
(methodInvocation.getMethod().getName().equals("buscarCliente")){
salida  =
gestorCache.buscarCliente((Integer)methodInvocation.getArguments()[0
]);
```

```
      if (salida == null)
            salida = methodInvocation.proceed();
        }

      }
      return salida;
}
```

- Búsqueda de conjuntos. Solo se podrá buscar siempre y cuando se cumpla la condición “Provincia in (‘Madrid’,’Barcelona’)”. En caso contrario hay que realizar la búsqueda en base de datos.

- Inserción/Borrado/Modificación (reflejando la operación en base de datos).

Puede que existan más campos que puedan generar este caso pero es recomendable solo utilizar un criterio (un campo) para plantear la cache. Si por ejemplo se da este caso, en la misma tabla con el campo provincia antes mencionado:

En la tabla “Cliente” también existe el campo que es “Edad”; dicho campo indica la edad del Cliente. El número de clientes con una edad de 30-40 años nunca superan los n elementos que si son un número de elementos que se pueden cachear y suponen la mayoría de accesos a la tabla.

Se puede estar tentado de implementar una cache para cada campo pero es desaconsejable por el gran coste de sincronización.

4.3.3 Caso tradicional

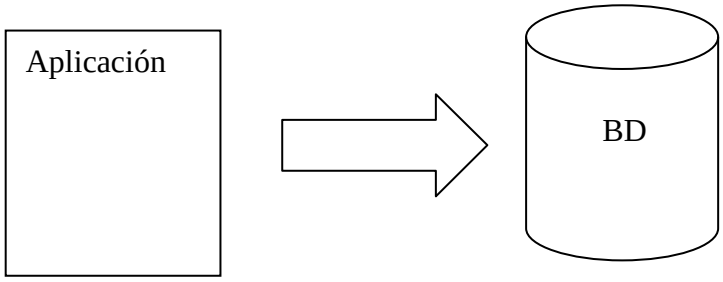
No existe nada parecido a los dos campos anteriores. En este caso lo que se debe hacer es seguir alguna de las técnicas tradicionales para guardar elementos en cache. Tales como FIFO (primero en entrar primero en salir), LIFO (ultimo en entrar primero en salir), etc…

Este caso permite las siguientes operaciones:

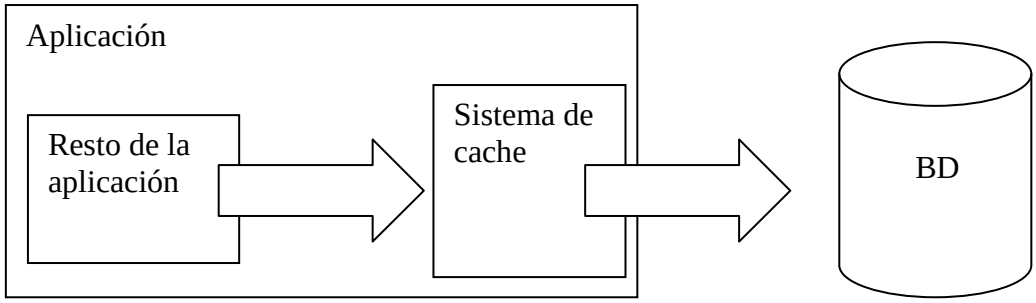
- Búsqueda unitaria. Igual que en el caso anterior.
- Inserción (reflejando la operación en base de datos).

4.4 Mejora de la propia aplicación

Si se implanta cache para manejar el acceso a los datos de la aplicación hemos pasado de un modelo de modificación directa de base de datos:



A un modelo en el que el sistema de cache decide cuando actualizar:



La flecha simboliza cualquier operación de acceso/modificación de base de datos.

Teniendo en cuenta este cambio de filosofía se pueden introducir modificaciones para mejorar todavía más el rendimiento.

4.4.1 Actualización de BD asíncrona

Si se ha de insertar, modificar o borrar datos de una tabla con cache completa los pasos lógicos son: primero reflejar en cache y después reflejar en base de datos. Pero si el dato ya está en cache realmente da igual cuando se realice la actualización de la base de datos por lo que se puede hacer de forma asíncrona para favorecer el rendimiento.

La clase AltaCliente tiene un método donde se da de alta un cliente y otro método en el que se busca un cliente de la manera siguiente:

```
public class AltaCliente {

    /**
     * Conjunto de funcionalidades en base de datos.
     */
    private Dao dao;
```

```
/**
     * Dar de alta un cliente.
     *
     * @param cliente datos del cliente a dar de alta.
     */
    public void darAltaCliente (Cliente cliente){

        dao.insertar(cliente);

    }

    /**
     * Buscar un cliente.
     *
     * @param cliente datos de busqueda del cliente.
     * @return cliente que cumple con los criterios de busqueda.
     */
    public Cliente buscarCliente (Cliente cliente){

        return dao.buscar(cliente);
    }
}
```

Se le implementa un sistema de cache:

```
    public class AltaCliente {

    /**
     * Conjunto de funcionalidades en base de datos.
     */
    private Dao dao;

 /**
     * Sistema de cacheo de datos.
     */
    private ManagerCache cache;


    /**
     * Dar de alta un cliente.
     *
     * @param cliente datos del cliente a dar de alta.
     */
    public void darAltaCliente (Cliente cliente){

        if (cache.estaFuncionando())
            cache.insertar(cliente);
        dao.insertar(cliente);

    }
```

```
/**
     * Buscar un cliente.
     *
     * @param cliente datos de busqueda del cliente.
     * @return cliente que cumple con los criterios de busqueda.
     */
    public Cliente buscarCliente (Cliente cliente){
        if (cache.estaFuncionando())

              return cache.buscar(cliente);
        else
              return dao.buscar(cliente);
    }
}
```

Se ha implementado lo más lógico. Al dar de alta se guarda el Cliente en el sistema de cache y secuencialmente en la base de datos, pero en realidad no importa que se guarde secuencialmente ya que la información ya está en el sistema de cache. Por lo que si se lanza la sentencia “dao.insertar(cliente);” en otro hilo de ejecución se puede ganar mucho rendimiento sobre todo si la inserción es muy pesada:

```
public class AltaCliente {
…
public void darAltaCliente (Cliente cliente){

        if (cache.estaFuncionando()){
            cache.insertar(cliente);
            Lanzar lanzar = new Lanzar(dao,cliente);
            lanzar.start();
        }else
            dao.insertar(cliente);
    }
….
```

```
public class Lanzar extends Thread{
    /**
     * Conjunto de funcionalidades en base de datos.
     */
    private Dao dao;

     /**
     * Objeto a guardar.
     */
    private Cliente cliente;

    Lanzar(Dao dao, Cliente cliente) {
        this.dao = dao;
        this.cliente = cliente;
    }

    @Override
    public void run() {
        dao.insertar(cliente);
    }
}
```

5. Conclusión

Siendo francos, un sistema de cache presenta muchas barreras a la hora de ser utilizado en una aplicación:

- Es un código que no aporta ninguna funcionalidad nueva.
- Es otra capa en la aplicación que puede dar fallos por lo que aumenta el nivel de pruebas.
- La filosofía de cacheo no es muy tan intuitiva y manejable como es el hecho de utilizar solo lenguaje Java llano o frameworks como Swing.

Pero en la actualidad existen tecnologías como Spring y AOP que permiten saltar dichas barreras. Entre las muchas ventajas que te ofrece el framework de Spring, está la transparencia y la claridad con la que se puede "incluir" o "quitar" una implementación de un sistema de cache. La programación orientada aspectos es perfecta para situaciones de posproducción de código como:

- Incluir un sistema de cache para mejorar el rendimiento de la aplicación.
- Resolución de incidencias sin modificar código original.
- Ampliar funcionalidad sin modificar código original.

Salvadas las barreras, cabe preguntar ¿El esfuerzo merece la pena? SIEMPRE MERECE LA PENA. Que la aplicación vaya más rápido es siempre un valor añadido para el cliente.

En el mundo del desarrollo de software actual es difícil encontrar un profesional al que no le haya pasado el caso en el que ha desarrollado una aplicación muy compleja, con muchas tecnologías, muy grande, etc.. y cuando toda funciona entonces el usuario ve una demo y dice:" Esto no va! … am ahora" ó "Va muy lento, ¿no?". Ya es posible evitar esta situación con las tecnologías actuales por lo que si al final del desarrollo (o en una fase madura del desarrollo de la aplicación) se observa que la aplicación va muy lenta no se la presentes al cliente como definitiva, **IMPLEMENTA UN SISTEMA DE CACHE**.

6. Referencias

- http://ehcache.org/

Página oficial de la implementación de cache EHCACHE.

- http://www.hibernate.org/

Página oficial de la implementación de ORM y de JPA

- http://www.springsource.org/

Página oficial del framework Spring.

- http://www.eclipse.org/aspectj/

Página oficial del framework AspectJ.

- http://www.java2s.com

Estupenda pagina de ejemplos cortos de código fuente en JAVA.