Bases de Datos Avanzadas

Técnicas de Indexación para las Bases de Datos Orientados a Objetos

Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle
Área de Lenguajes y Sistemas Informáticos. Departamento de Informática. Universidad de Oviedo

<{belen,cueva}@pinon.ccu.uniovi.es>

Resumen: *en este artículo se presenta una clasificación de las técnicas de indexación para las bases de datos OO atendiendo a las características del modelo de objetos a las que pretenden dar soporte (jerarquías de herencia, de agregación e invocación de métodos), haciendo una breve descripción de algunas de dichas técnicas: SC, CH-Tree, H-Tree, Nested, Path, MultiIndex, Nested Inherited y Method Materialization.*

1. Introducción

Los sistemas de gestión de bases de datos orientadas a objetos (**SGBDOO**) deben su aparición principalmente, al surgimiento de nuevos tipos de aplicaciones para las que el modelo de objetos representaba mejor la semántica de la nueva información a almacenar. Los **SGBDOO** ya clásicos como ObjectStore, O2, Poet, Versant y GemStone, proporcionan soporte para lenguajes de consulta declarativos de alto nivel, ya que como se demostró en el modelo relacional, dichos lenguajes reducían substancialmente el tiempo de desarrollo de las aplicaciones. De hecho, el propio intento de estandarización lanzado por **ODMG** [1] propone un lenguaje de consulta (**OQL**).

En la actualidad, coexistiendo con estos SGBDOO ya clásicos se encuentran los motores de almacenamiento persistente como complemento a los lenguajes de programación OO (C++ y Java, principalmente), tales como PSE Pro para Java [2] y Jeevan [3], que permiten también de alguna forma que se realicen consultas a la base de datos.

De esto se deduce que es necesario incluir en estos productos (tanto los clásicos como en los más modernos) mecanismos, como los índices, que permitan acelerar el procesamiento de las consultas.

1.1. Lenguajes de Consulta OO

Si nos centramos en los lenguajes de consulta orientados a objetos, conviene resaltar tres características que vienen directamente impuestas por el modelo de objetos (y que marcarán claramente las diferencias con los lenguajes de consulta relacionales):
a) ***El mecanismo de la herencia (Jerarquías de Herencia).*** La herencia provoca que una instancia de una clase sea también una instancia de su superclase. Esto implica que, el ámbito de acceso de una consulta sobre una clase en general incluye a todas sus subclases, a menos que se especifique lo contrario.
b) ***Predicados con atributos complejos anidados (Jerarquías de Agregación)***. Mientras que en el modelo relacional los valores de los atributos se restringen a tipos primitivos simples, en el modelo orientado a objetos el valor del atributo de un objeto puede ser un objeto o conjunto de objetos. Esto provoca que las condiciones de búsqueda en una consulta sobre una clase se puedan seguir expresando de la forma <atributo operador valor>, al igual que en el modelo relacional, pero con la diferencia básica de que el atributo puede ser un atributo anidado de la clase.
c) ***Predicados con invocación de métodos***. En el modelo de objetos los métodos definen el comportamiento de los objetos y, al igual que en el predicado de una consulta puede aparecer un atributo, también puede aparecer la invocación de un método.

1.2. Clasificación de las Técnicas de Indexación en OO

Las características anteriormente mencionadas exigen técnicas de indexación que permitan un procesamiento eficiente de las consultas bajo estas condiciones. Así, son muchas las técnicas de indexación en orientación a objetos que se han propuesto y que clásicamente [4] se pueden clasificar en:
a) ***Estructurales***. Se basan en los atributos de los objetos. Estas técnicas son muy importantes porque la mayoría de los lenguajes de consulta orientados a objetos permiten consultar mediante predicados basados en atributos de objetos. A su vez se pueden clasificar en:
- Técnicas que proporcionan soporte para consultas basadas en la Jerarquía de Herencia. Ejemplos de los esquemas investigados en esta categoría son SC [5], CH-Tree [5], H-Tree [6], Class Division [7], hcC-Tree [8], etc.
- Técnicas que proporcionan soporte para predicados anidados, es decir, que soportan la Jerarquía de Agregación. Ejemplos de los índices de esta categoría [9] son, entre otros, Nested, Path y Multiindex.
- Técnicas que soportan tanto la Jerarquía de Agregación como la Jerarquía de Herencia. Nested Inherited [4] es un ejemplo de esta categoría.

b) ***De Comportamiento***. Proporcionan una ejecución eficiente para consultas que contienen invocación de métodos. La **materialización de métodos** (*method materialization* [10]) es una de dichas técnicas. En este campo no existe una proliferación de técnicas tan grande como en los anteriores.

En la sección 2 se describen algunas técnicas que proporcionan soporte para consultas centradas en la jerarquía de herencia. Las consultas centradas en la jerarquía de agregación pueden emplear técnicas como las descritas en la sección 3. Cuando en una consulta se vean implicados

objetos anidados y jerarquías de herencia, y se desee realizar una búsqueda en un único índice, se puede emplear la técnica descrita en la sección 4. Finalmente, en la sección 5 se describe una técnica que pretende acelerar el procesamiento de consultas que permiten la invocación de métodos.

2. Técnicas basadas en la Jerarquía de Herencia

Estas técnicas se basan en la idea de que una instancia de una subclase es también una instancia de la superclase. Como resultado, el ámbito de acceso de una consulta contra una clase generalmente incluye no sólo sus instancias sino también las de todas sus subclases. Con el fin de soportar las relaciones de subclase-superclase eficientemente, el índice debe cumplir dos objetivos:
- la recuperación eficiente de instancias de una clase simple;
- la recuperación eficiente de instancias de clases en una jerarquía de clases.

2.1. *Single Class* (SC)

La técnica *Single Class* fue una de los primeras en emplearse [5]. En esta técnica, la creación de un índice para un atributo de un objeto requiere la construcción de un árbol B+ para cada clase en la jerarquía indexada. Es la más tradicional, ya que su estructura es la de un árbol B+ típico, y además, mantiene la idea del modelo relacional de un índice para cada relación y atributo indexado.

2.2. *CH-Tree*

Kim et al. [5] proponen un esquema llamado "Árbol de Jerarquía de Clases" (*Class Hierarchy Tree- CH Tree*) que se basa en mantener un único árbol índice para todas las clases de una jerarquía de clases. El *CH-Tree* indexa una jerarquía de clases sobre un atributo común, típicamente uno de los atributos de la superclase, sobre una estructura de un árbol B+.

2.2.1. Estructura

La estructura del *CH-Tree* está basada en los árboles B+, y de hecho, el nodo interno es similar al de éstos. La diferencia está en los nodos hoja. Éstos contienen (**figura 1**):
· Para cada clase en la jerarquía, el número de elementos almacenados en la lista de identificadores de objetos (**IDOs**) que corresponden a los objetos que contienen el **valor-clave** en el atributo indexado
· La lista de IDOs

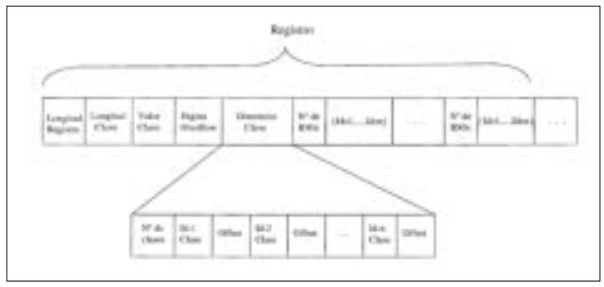

Figura 1: Nodo hoja de un CH-Tree

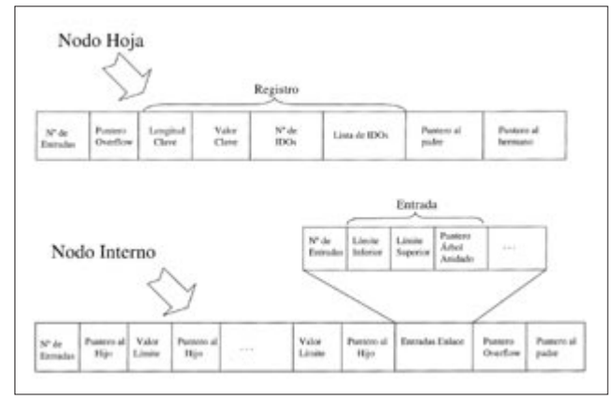

Figura 2: Estructura de un H-Tree

· Un **directorio clave** que almacena el número de clases que contienen objetos con el valor-clave en el atributo indexado, y, para cada clase, el identificador de la misma y el desplazamiento en el registro índice, que indica donde encontrar la lista de IDOs de los objetos.

Es decir, el nodo hoja agrupa la lista de IDOs para un valor-clave en términos de las clases a las que pertenecen.

La búsqueda en un *CH-Tree* es similar a la de los árboles B+, de tal manera que, cuando el nodo hoja es localizado, se comparan los valores clave almacenados en el registro índice con la clave buscada. Si se encuentra, entonces, si en la consulta se referencia la jerarquía de clases completa, se devuelven todos los IDOs que aparecen en el registro índice. Por el contrario, si en la consulta se referencia sólo una clase, se consulta el directorio clave del registro índice para localizar los IDOs asociados con la clase. Esta técnica es empleada, por ejemplo, en el sistema ORION [11].

2.3. *H-Tree*

Representan una alternativa al *CH-Tree* propuesta en [6]. Se basan también en los árboles B+, y pueden verse como los sucesores de los índices **SC**. Esta técnica, se basa en mantener un **H-Tree** para cada clase de una jerarquía de clases, y estos **H-Trees** se anidan de acuerdo a sus relaciones de superclase-subclase. Cuando se indexa un atributo, el *H-Tree* de la clase raíz de una jerarquía de clases, se anida con los *H-Trees* de todas sus subclases inmediatas, y los *H-Trees* de sus subclases se anidan con los *H-Trees* de sus respectivas subclases, y así sucesivamente. Indexando de esta manera se forma una jerarquía de árboles de índices.

2.3.1. Estructura

La estructura del *H-Tree* presenta variaciones tanto en el nodo interno como en el nodo hoja, con relación a los árboles B+ (**figura 2**). El nodo hoja contiene un contador que índica el número de IDOs que hay en la lista de identificadores de objetos cuyo valor en el atributo indexado es el valor-clave, y la propia lista de IDOs.

El nodo interno, aparte de los valores clave discriminantes y los punteros a los nodos hijos, almacena punteros que apuntan a subárboles de *H-Trees* anidados. Para reducir el

recorrido innecesario del subárbol anidado, los valores máximos y mínimos del subárbol anidado se mantienen junto con el puntero a dicho subárbol.

El anidamiento de los *H-Trees* evita la búsqueda en cada *H-Tree* cuando se consultan un número de clases de la jerarquía [12]. Cuando se busca en los árboles de una clase y sus subclases, se hace una búsqueda completa en el *H-Tree* de la superclase, y una búsqueda parcial en los *H-Trees* de las subclases. Esta es la principal ventaja con relación a los índices SC. Además, un *H-Tree* puede ser accedido, independientemente del *H-Tree* de su superclase, ya que la clase consultada no tiene porque ser la clase raíz de la jerarquía de clases, y por tanto, buscar instancias dentro de una subjerarquía de clases puede comenzar en cualquier clase siempre que esté indexada por el mismo atributo. El mayor inconveniente de esta estructura es la dificultad para dinamizarla.

2.4. Comparación : SC, *CH-Tree* y *H-Tree*

Para consultas realizadas únicamente contra una clase, el SC se comporta mejor que el *CH-Tree*. Si por el contrario, se ven implicadas todas las clases de la jerarquía (o al menos dos clases de la misma) el *CH-Tree* se comporta mejor que un conjunto de SC. En cuanto al tamaño de los índices generados, no hay estudios suficientes que permitan concluir que uno sea mejor que otro.

Estudios realizados demuestran también que el *H-Tree* se comporta mejor que el *CH-Tree* para consultas de recuperación, especialmente cuando un pequeño número de clases en la jerarquía son referenciados por la consulta, ya que el *CH-Tree* emplea la misma estrategia para la recuperación de datos de una clase simple que de una jerarquía de clases.

3. Técnicas basadas en la Jerarquía de Agregación

Estas técnicas se basan en la idea de que las consultas en orientación a objetos soportan predicados anidados. Para soportar dichos predicados, los lenguajes de consulta generalmente proporcionan alguna forma de expresiones de camino.

Camino

Un camino es una rama en una jerarquía de agregación o, dicho formalmente, un camino P para una jerarquía de clases H se define como C(1).A(1).A(2)...A(n), donde C(1)

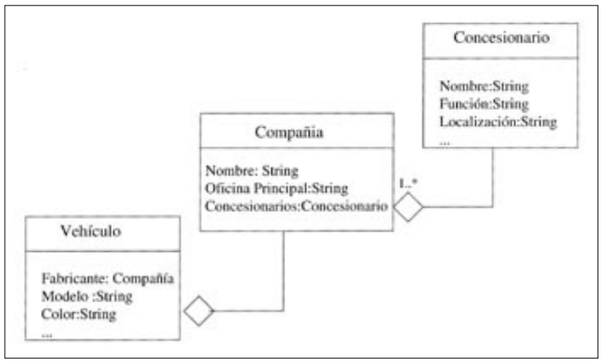

Figura 3 : Jerarquía de clases de ejemplo

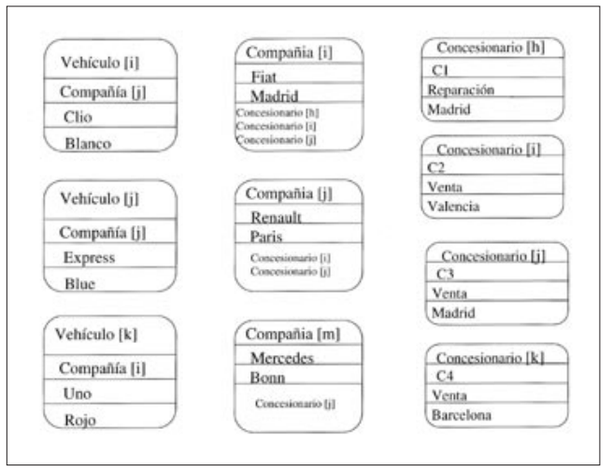

Figura 4 :Instancias para la jerarquía de la Figura 3

es el nombre de una clase en H, A(1) es un atributo de la clase C(1), y A(i) es un atributo de una clase C(i) en H, tal que C(i) es el dominio del atributo A(i-1) de la clase C(i-1), $1< i \leq n$. Un ejemplo de camino para el esquema de la **figura 3** puede ser P1=*Vehículo.Fabricante.Nombre*.

Instanciación de un camino

Dado un camino P = C(1).A(1).A(2)...A(n), una **instanciación** de P se define como una secuencia de n+1 objetos, denotados como O(1).O(2)...O(n+1), donde, O(1) es una instancia de la clase C(1), y O(i) es el valor de un atributo A(i-1) del objeto O(i-1), $1< i \leq n+1$. Por ejemplo, para el camino P1 una instanciación sería *Vehículo[i]. Compañía[j].Renault*.

Instanciación Parcial

Dado un camino P= C(1).A(1).A(2)...A(n), una **instanciación parcial** de P es definida como una secuencia de objetos O(1).O(2)...O(j), $j< n+1$, donde O(1) es una instancia de una clase C(k) en *Class* (P) para k=n-j+2, y O(i) es el valor del atributo A(i-1) del objeto O(i-1), $1< i \leq j$, y siendo *Class*(P) = C(1) $\lceil$ {C(i) tal que C(i) es el dominio del atributo A(i-1) de la clase C(i-1), $1 < i \leq n$}. Atendiendo a los datos de la **figura 4**, una instanciación parcial del camino P1 sería *Compañía[i].Fiat*.

Dada una instanciación parcial p= O(1).O(2)...O(j) de P, p es no redundante, si no existe una instanciación p'=O'(1).O'(2)...O'(k), k>j, tal que O(i) = O'(k-j+i), $1\leq i \leq j$. Una instanciación parcial no **redundante** para P1 sería *Compañía[m].Mercedes*.

3.1.Índice Path (PX)

Dado un camino P= C(1).A(1).A(2)...A(n), un índice *Path* (PX) sobre P es definido como un conjunto de pares (O,S), donde O es el **valor clave**, y S={π<j-1>(p(i)) tal que, p(i)=O(1).O(2)...O(j) es una instanciación no redundante (parcial o no) de P, y O(j) = O}. π<j-1>(p(i)) denota la proyección de p(i) sobre los primeros j-1 elementos. De esto se deduce que, un índice PX almacena todos los objetos que finalizan con esa clave.

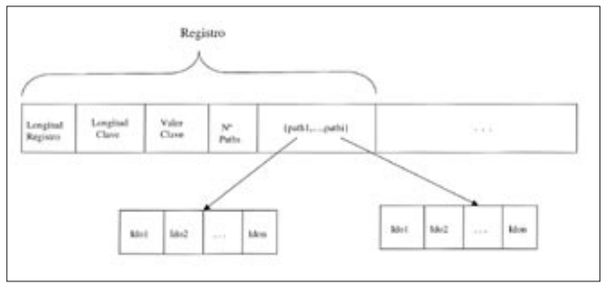

Figura 5 : Nodo hoja para un Indice Path

Por ejemplo, para el camino P1 el índice PX contendrá los siguientes pares:
· (Fiat,{Vehículo[k].Compañía[i]})
·(Renault,{[Vehículo[i].Compañía[j],
· Vehículo[j].Compañía[j]})
· (Mercedes,{Compañía[m]})

Un PX almacena instanciaciones de caminos, y como se puede observar, puede ser empleado para evaluar predicados anidados sobre todas las clases a lo largo del camino.

3.1.1. Estructura

La estructura de datos base para este índice puede ser un árbol B, al igual que para los Nested y Multiindex, y de hecho el formato de los nodos internos es idéntico en los tres casos. La variación está en el nodo hoja. En el caso del índice PX, el nodo hoja (**figura 5**) contiene, entre otra información, el número de elementos en la lista de instanciaciones que tienen como objeto final el **valor clave** y la propia lista de instanciaciones.

3.2. Índice Nested (NX)

Dado un camino P= C(1).A(1).A(2)...A(n), un índice *Nested* (NX) sobre P se define como un conjunto de pares (O,S), donde O es el **valor clave**, y S es el conjunto de IDOs de objetos de C(1) tales que O y cada IDO en S, aparecen en la misma instancia del camino. Es decir, un índice NX proporciona una asociación directa entre el primer y el último objeto de un camino. A diferencia del índice *Path* que almacena todos los objetos que finalizan con ese valor clave, en un índice *Nested*, sólo se almacenan los objetos iniciales de las instanciaciones de los caminos.

Por ejemplo para el camino P1, el NX contendrá los siguientes pares:
· (Fiat,{Vehículo[k]})
· (Renault,{Vehículo[i],Vehículo[j]})

Como se puede observar cuando n=1, el NX y el PX son idénticos y son los índices empleados en la mayoría de los SGBD relacionales.

3.2.1. Estructura

El nodo hoja para el índice *Nested* es similar al del Multiindex, y simplemente almacena la longitud del registro, la longitud de la clave, el valor clave, el número de elementos en la lista de IDOs de los objetos que contienen el valor clave en el atributo indexado, y la lista de IDOs.

3.3. Multiindex (MX)

Dado un camino P= C(1).A(1).A(2)...A(n), un *Multiindex* (MX) se define como un conjunto de n *índices Nested* (NX), NX.1, NX.2,... NX.n, donde NX.i es un índice *Nested* definido sobre el camino C(i).A(i), $1 \leq i \leq n$. Por ejemplo, para el camino P1 el Multiindex consta de dos índices *Nested*. El primero, sobre el subcamino Vehículo.Fabricante, y contendrá los siguientes pares:
· (Compañía[i],{Vehículo[k]})
· (Compañía[j],{Vehículo[i], Vehículo[j]})

El segundo índice es sobre el subcamino Compañía.Nombre, y contendrá los siguientes pares:
· (Fiat,{Compañía[i]})
· (Renault,{Compañía[j]})
· (Mercedes,{Compañía[m]}).
Esta técnica es empleada en el sistema GemStone.

La razón de dividir un camino en subcaminos, es principalmente, para reducir los costos de actualización del NX o del PX, pero permitiendo a la vez una recuperación eficiente.

En [13] se propone un algoritmo que determina si un camino debe ser dividido en subcaminos y también a qué subcaminos se les debe asignar un índice, así como la organización (NX, PX, etc.) que éste debería tener.

3.4. Comparación : NX, PX y MX

Atendiendo a los experimentos realizados y expuestos en [9], se puede concluir que en cuanto a requerimientos de espacio, el NX siempre tiene los requerimientos más bajos. El PX requiere siempre más espacio que el MX, excepto cuando hay pocas referencias compartidas entre objetos, o bien no hay ninguna. Esto es debido a que el mismo IDO puede aparecer en muchas instancias de camino, ocasionando que ese IDO sea almacenado muchas veces.

Si se examina el rendimiento con relación a consultas de recuperación, el NX es el que mejor se comporta, ya que el PX tiene un gran tamaño, pero este último se comporta bastante mejor que el MX, ya que éste necesita acceder a varios índices. Para consultas de actualización, el que mejor se comporta es el MX, seguido del PX y finalmente el NX. Para concluir, se puede decir que teniendo en cuenta las diferentes clases de operaciones (recuperación, actualización, inserción y borrado) el PX se comporta mejor que las otras dos estructuras.

4. Técnicas basadas en la Jerarquía de Herencia y de Agregación

Estas técnicas de indexación proporcionan soporte integrado para consultas que implican atributos anidados de objetos y jerarquías de herencia. Es decir, permiten que una consulta que contiene un predicado sobre un atributo anidado e implica varias clases en una jerarquía de herencia dada, se solucione con una búsqueda en un único índice.

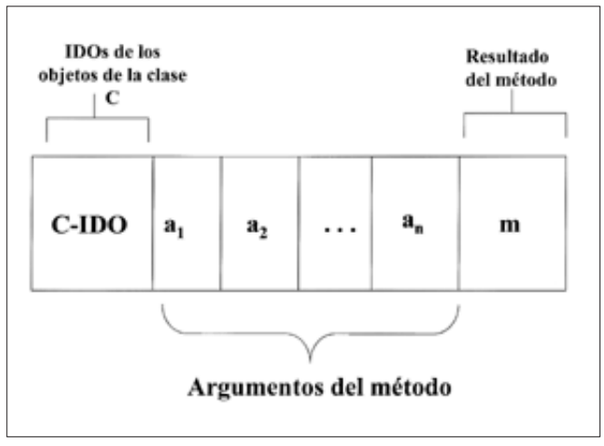

Figura 6 :Formato de la tabla que almacena los datos materializados

4.1. *Nested Inherited*

Dado un camino P= C(1).A(1).A(2)...A(n), un índice *Nested Inherited* (NIX) asocia el valor v del atributo A(n) con los IDOs de las instancias de cada clase en el ámbito de P que tienen v como valor del atributo anidado A(n). Entendiéndose por ámbito de un camino el conjunto de todas las clases a lo largo de un camino y todas sus subclases.

4.1.1. Estructura

El formato de un nodo no hoja tiene una estructura similar a los índices tradicionales basados en árboles B+. Sin embargo, la organización del registro de un nodo hoja es similar al del CH-Tree, en el sentido de que hay un directorio dentro del registro del nodo hoja que asocia a cada clase el desplazamiento, dentro del registro, donde son almacenados los IDOs de las instancias de la clase. Sin embargo, un NIX puede ser empleado para consultar todas las jerarquías de clases encontradas a lo largo de un camino, mientras que un CH-Tree sólo permite consultar una única jerarquía de clases. Para más detalle sobre el nodo hoja ver [4].

La organización del índice consta de dos índices. El primero, llamado **indice primario**, que es indexado sobre los valores del atributo A(n). Asocia con un valor v de A(n), el conjunto de IDOs de las instancias de todas las clases que tienen v como valor del atributo anidado A(n). El segundo índice, llamado **índice secundario**, tiene IDOs como claves de indexación. Asocia con el IDO de un objeto la lista de IDOs de sus padres. Los registros del nodo hoja en el índice primario contienen punteros a los registros del nodo hoja del índice auxiliar, y viceversa. El índice primario es utilizado para las operaciones de recuperación, mientras que el índice secundario es empleado, básicamente, para determinar todos los registros primarios donde son almacenados los IDOs de una instancia dada, con el fin de realizar eficientemente las operaciones de inserción y borrado.

5. Materialización de Métodos

El uso de métodos es importante porque define el comportamiento de los objetos, sin embargo, tiene dos efectos negativos desde el punto de vista del procesamiento y optimización de consultas:

- Es caro invocar un método, ya que eso ocasiona la ejecución de un programa y si esto tiene que realizarse sobre un conjunto grande de objetos ralentizará el procesamiento de la consulta.
- La implementación de cada método está oculta (por la encapsulación del modelo OO), por lo que es muy difícil estimar el coste de su invocación en una consulta.

La **materialización de métodos** es una técnica que pretende aliviar estos problemas. La idea básica consiste en calcular los resultados de los métodos accedidos frecuentemente, 'a priori', y almacenar los resultados obtenidos para emplearlos más tarde.

Estructura

El resultado materializado de un método puede ser almacenado en una tabla con la estructura mostrada en la **figura 6**. El tamaño de esta tabla puede llegar a ser muy grande, con lo que se pueden crear índices para acelerar la velocidad de acceso a la misma. Cuando varios métodos para la misma clase tengan el mismo conjunto de argumentos, los resultados materializados de estos métodos pueden ser almacenados en la misma tabla, con lo que se necesita menos espacio, y si además estos métodos son referenciados en la misma consulta, sólo sería necesario acceder a una tabla para evaluar estos métodos, lo que se traduciría en un coste de procesamiento más bajo. Si los resultados de cada método referenciado en una consulta han sido precalculados, será mucho más fácil para el optimizador de consultas generar un buen plan de ejecución para la consulta.

Eficiencia en las consultas de recuperación y Mayor costo en las consultas de actualización

La materialización de métodos permite un procesamiento más eficiente de las consultas de recuperación, pero supone un costo más alto para las consultas de actualización, ya que una operación de actualización puede ocasionar que el resultado precalculado llegue a ser inconsistente con los datos actualizados.

Una cuestión importante a tener en cuenta en la materialización de métodos es el manejo eficiente de este problema de inconsistencia. Así, cuando se borra un objeto, todas las entradas materializadas que usan ese objeto deben ser borradas y cuando un objeto existente es modificado todos los resultados materializados que usen el objeto modificado necesitan ser recalculados. Este proceso puede ser inmediato, o retrasado hasta que el resultado de la instancia del método sea necesario para evaluar la consulta (*lazy materialization*). En [10] se presentan además varias técnicas que permiten reducir el costo de mantenimiento de los métodos materializados.

6. Conclusiones

Los mecanismos de indexación son extremadamente importantes para el procesamiento de consultas en BDOO. El objetivo de este artículo era describir brevemente algunas de las técnicas de indexación en OO ya clásicas, clasificadas en función de las características del modelo de objetos a las que pretenden dar soporte (agregación, herencia, métodos). No

	JH	JA	IM
SC, CH-Tree, H-Tree, Class Division, ...	**X**		
Nested, Path, Multiindex, ...		**X**	
Nested Inherited, ...	**X**	**X**	
Method Materialization,...			**X**
JH- Jerarquía de Herencia ; JA- Jerarquía de Agregación; IM- Invocación de métodos			

Tabla 1: Clasificación de las técnicas de Indexación en OO en función de las características que soportan

obstante, al ser la indexación un tema clave dentro de las bases de datos, existen más recientes investigaciones sobre el tema que no son abarcadas en este artículo pero a las que se puede acceder desde [14].

De lo expuesto en el artículo se deduce que, a pesar de que la proliferación en técnicas de indexación en OO es grande, no se puede concluir que una de ellas sea mejor que las demás en todas las situaciones, o al menos bajo las condiciones en las que han sido probadas.

7. Referencias

[1] Cattell R., Barry D., Bartels D. *The Object Database Standard: ODMG 2.0.* Morgan Kauffman, 1997.
[2] ObjectStore PSE and PSE Pro for Java. *User Guide*. *http://www.odi.com http://www.odi.com*, Marzo 1999.
[3] Jeevan User's Guide. *http://www.w3apps.com http://www.w3apps.com*, Marzo 1999.
[4] Bertino E. y Foscoli P. Index Organizations for Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering*. Vol.7, 1995.
[5] Kim W., Kim K.C., Dale A. Indexing Techniques for Object-Oriented Databases. En W. Kim y F.H. Lochovsky (ed): *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley, 1989.
[6] Chin C., Chin B., Lu H. *H-trees: A Dinamic Associative Search Index for OODB.* ACM SIGMOD, 1992.
[7] Ramaswamy S. y Kanellakis C. *OODB Indexing by Class Division*. ACM SIGMOD, 1995.
[8] Sreenath B. y Seshadri S. *The hcC-Tree: An Efficient Index Structure for Object Oriented Databases*. International Conference on VLDB, 1994.
[9] Bertino E. y Kim W. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering.* Vol.1 nº2, 1989.
[10] Kemper A., Kilger C. y Moerkotte G. Function Materialization in Object Bases: Design, Realization, and Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 1994.
[11] Kim W.,Garza J.F., Ballou N and Woelk D. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, Marzo, 1990.
[12] Chin B., Han J., Lu H. y Lee K. Index nesting - an efficient approach to indexing in object-oriented databases. *VLDB Journal*, 1996.
[13] Bertino E. Index Configuration in Object Oriented Databases. *VLDB Journal* ,3, 1994.
[14] SGBDOO para Oviedo3.*http://www.uniovi.es/~oviedo3/belen/bdoviedo3.html*, Junio 1999.