Una propuesta para el manejo de recursión en SQL

Francisco Javier Moreno Arboleda [1]

Jaime Alberto Guzmán Luna [2]

RESUMEN

En este artículo se propone un método para la concepción de consultas recursivas en SQL (Structured Query Language). Este método constituye una alternativa que puede ser considerada tanto desde el punto de vista de la optimización de consultas como de Sistemas de Gestión de Bases de Datos que no disponen de operadores especializados para soportar tales tipos de consultas.

Palabras clave: árboles, consultas recursivas, jerarquías, lenguajes de consulta, SQL, recursión.

A proposal to the treatment of recursion in SQL

ABSTRACT

In this article an alternative method is proposed in order to pose recursive queries in SQL (Structured Query Language). This method is an alternative that can be considered from the point of view of query optimization as well as Database Management Systems that do not have specialized operators for such types of queries.

Key words: trees, recursive queries, hierarchies, query languages, SQL, recursion.

1. INTRODUCCIÓN

Desde 1996 con la inclusión de la opción procedimental o PSM (Persistent Stored Modules) al SQL [1] éste se vuelve computacionalmente completo, gracias a las estructuras clásicas de programación: secuencia, decisión e iteración. Se puede distinguir entre el SQL "puro" y el SQL + PSM. Los PSM se han incorporado en diversos Sistemas de Gestión de Bases de Datos (SGBD). En Oracle por ejemplo la opción procedimental se llama PL/SQL y en SQL Server TSQL.

Con la incorporación de la opción procedimental es posible resolver prácticamente cualquier consulta por medio de SQL debido a que las funciones, que se programan en PSM, pueden ser invocadas desde las consultas. Estas funciones pueden ser tan complejas como se requiera, incluso pueden ser recursivas. Por otro lado el aspecto de la optimización de consultas en los SGBDs es un problema prioritario y en constante investigación.

Aunque la recursión es una técnica poderosa para dar solución a determinados problemas tiene como contraparte su enorme costo computacional. Varios SGBDs poseen operadores especializados para lograr la recursión en SQL puro, por ejemplo en Oracle, *CONNECT BY* [2] y en DB2 *WITH* [3] (véase la Sección 3). Con el mismo fin, desde el SQL estándar 1999 [4] se ha adicionado un operador [5], con el mismo nombre que en DB2; aunque la versión de este operador en DB2 no posee todas las características que establece el SQL estándar [3]. Por otro lado, Libkin [6] demuestra que existen consultas recursivas que definitivamente no pueden ser expresadas mediante el SQL estándar anterior al SQL-99 y realiza un análisis sobre la expresividad de SQL.

Date [7] propone un operador *EXPLODE* el cual puede ser incorporado en un lenguaje relacional de consulta para solucionar consultas tipo "explosión de partes" (véase la Sección 2). Agrawal [8] propone extender el álgebra relacional con un operador recursivo denominado *ALPHA*, Ahad [9] propone un lenguaje especializado basado en álgebra relacional denominado RQL (Recursive Query Language) y Shan [10] propone extender el álgebra relacional mediante un operador *FIXPOINT* para soportar consultas recursivas.

Linnemann [11] aborda el problema de consultas recursivas pero para relaciones que no están en primera forma normal (relaciones que en la intersección de una fila y columna pueden tener múltiples valores).

Tillquist [12] propone los operadores *TCLOSE* (transitive closure), *COMPOUND*,

[1] Profesor de la Escuela de Sistemas de la Facultad de Minas, Universidad Nacional de Colombia.

[2] Profesor de la Escuela de Sistemas de la Facultad de Minas, Universidad Nacional de Colombia.

Artículo recibido en Abril de 2006, aprobado para publicación en Agosto de 2007

El método descrito no requiere extender el lenguaje SQL con operadores especializados

GROUP BY NODES y *GROUP BY LEAVES* para tratar consultas recursivas en QUEL [13] (un lenguaje de consulta similar a SQL). Los autores explican que dichos operadores pueden también ser propuestos para SQL. Jagadish [14] acude también al concepto de cierre transitivo [15] para abordar consultas recursivas en bases de datos pero orientadas a lenguajes tipo Prolog. Prolog es un lenguaje basado en reglas utilizado en el campo de la inteligencia artificial. En este tipo de lenguajes el planteamiento de expresiones recursivas es natural. Prolog se ha extendido para consultas en bases de datos, dicha extensión se conoce como Datalog, una detallada explicación de este lenguaje puede verse en Maier [16]. Koymen [17] propone un operador recursivo para SQL similar a como se trabaja en Datalog. Rosenthal [18] presenta algoritmos eficientes para la implementación de la operación cierre transitivo y como éstos pueden incorporarse en el optimizador de un SGBD. Introduce el concepto de *traversal recursions* el cual es una generalización del cierre transitivo y que permite mayor flexibilidad para el planteamiento de consultas recursivas.

James [19] propone operadores especializados para "tablas con árboles" y Biskup [20] propone extender SQL con un conjunto de operadores que faciliten la concepción de consultas para tablas "basadas en grafos". Cruz [21] propone un lenguaje especializado llamado G para la concepción de consultas recursivas. Tal y como lo exponen los autores, su lenguaje no pretende ser una alternativa para lenguajes de consulta relacionales, sino un lenguaje complementario en el cual las consultas recursivas son el objetivo.

Partiendo de la propuesta del SQL-99 para la concepción de consultas recursivas, Ordonez [22] trata los aspectos relativos a la optimización de tales consultas.

Todas las propuestas mencionadas conciben un lenguaje especializado para tratar consultas recursivas o proponen extender el SQL o el álgebra relacional con operadores especializados.

El método aquí planteado no requiere extender al SQL con un operador especializado. En la misma línea de trabajo se encuentran las siguientes propuestas. Celko [23][24] propone un método denominado "anidado de árboles", sin embargo es complejo manejar ciertas consultas en él, por ejemplo para hallar la altura del árbol (el cual se genera a raíz de una tabla que exhibe una relación recursiva, véase la Sección 2) se requiere una reunión de la tabla consigo misma más una operación de grupo (GROUP BY); en el método aquí propuesto no se requiere ni la reunión ni el agrupamiento (véase la consulta 4 en la subsección 3.3.1). Similarmente hallar los descendientes directos de un nodo implica en su método una consulta con reunión más una subconsulta correlacionada (confróntese con la consulta 2 en la subsección 3.3.1).

Moreau [25] presenta un método denominado *ruta posicional*. Posee similitudes con el método aquí propuesto, pero no maneja bien ciertas consultas ya que requiere subconsultas o reuniones que pueden ser evitadas (véase la consulta 3 en la subsección 3.3.1).

El método propuesto en este artículo para el planteamiento de consultas recursivas en SQL es una alternativa que puede ser contemplada tanto desde el punto de vista de optimización de consultas como de SGBDs que no posean operadores especializados para consultas recursivas o que aunque posean alguna versión de PSM o interactúen con un lenguaje que permita la recursión (Java, C++, C# etc.) su costo computacional sea alto.

El artículo se desarrolla así: en la Sección 2 se presenta un caso de estudio que ejemplifica una situación recursiva. En la Sección 3 se explica y ejemplifica el método propuesto para plantear consultas recursivas. En la Sección 4 se exponen algunas limitaciones del método y posibles formas de evitarlas. Finalmente, en la Sección 5 se presentan conclusiones y trabajos futuros.

2. CASO DE ESTUDIO

Las consultas recursivas pueden surgir en muchas situaciones: (i) requisitos u objetivos que poseen a su vez subrequisitos o subobjetivos, (ii) árboles genealógicos, (iii) descomposición estructural de una compañía en departamentos y éstos a su vez en sub-departamentos y (iv) composición de productos basados en otros productos, situación conocida como "explosión de partes" [7].

Sea la consulta "obtener todos los productos que componen a otro producto". Una solución impráctica [26] a este tipo de consultas es realizar una serie de reuniones de una tabla con-

sigo misma *n* veces, donde *n* indica el nivel de descomposición o profundidad deseado, es decir, si se desea, por ejemplo, en el caso de un árbol genealógico, determinar los bisnietos de un individuo, se debe realizar una triple reunión de la tabla consigo misma. Aparte de impráctico, porque el máximo nivel de profundidad se desconoce normalmente, el costo computacional de una *n* auto reunión es enorme.

Se plantea a continuación un método alternativo para abordar este tipo de consultas. Considérese la siguiente situación: supóngase que un cliente puede recomendar a muchos clientes y a su vez cada uno de éstos puede recomendar a otros y así sucesivamente. Un cliente no tiene necesariamente que haber sido recomendado por otro cliente pero en caso de que lo sea sólo puede ser recomendado por uno y sólo uno.

En la Figura 1 se muestra un modelo entidad relación que representa la situación descrita

Dicho modelo implantado de manera relacional produce el esquema mostrado en la Tabla I.

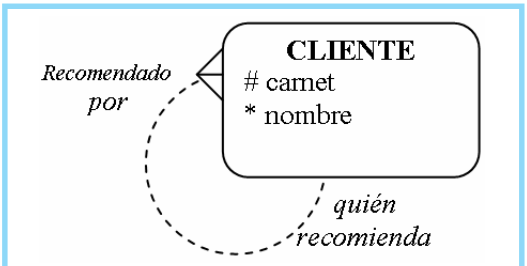

Figura 1. Modelo Entidad Relación del caso de estudio

Tabla I. Esquema de la Relación Cliente

Columnas	Características
Carnet	Clave Primaria
Nombre	Obligatorio
Recomendado_por	Clave foránea opcional hacia Cliente

En la Tabla II se presenta una muestra de la relación Cliente.

En la Figura 2 se muestra una representación arbórea de la jerarquía de recomendación existente en la relación Cliente.

3. MÉTODO PROPUESTO

Partiendo del caso presentado en la Sección 2, sea la consulta: ¿cuántos recomendados (directos e indirectos) posee el cliente con carnet

Tabla II. Relación Cliente

carnet	Nombre	recomendado_por
13	Ana	NULL
23	León	13
81	Carlos	13
12	Héctor	23
14	Juan	23
5	Lisa	81
6	Maria	81
33	Pedro	81
9	Luis	14
17	Dora	6
27	Iván	5
39	Rosa	27

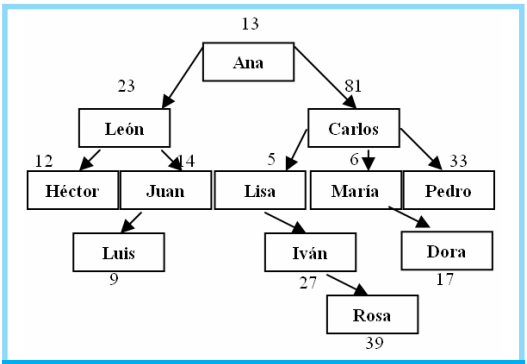

Figura 2. Representación jerárquica de las recomendaciones en la relación Cliente.

81 (Carlos)? En este caso la respuesta es 6 (véase la Figura 2). Dicha consulta se puede resolver haciendo uso de múltiples reuniones o usando el operador *CONNECT BY* en Oracle como se muestra a continuación:

```
SELECT COUNT(*)-1 AS TOTAL
FROM cliente
START WITH carnet = 81
CONNECT BY PRIOR carnet =
recomendado_por;
```

El operador *CONNECT BY* se puede acompañar de la función *level*, que permite obtener el nivel de profundidad de un nodo. Otras extensiones recientes para *CONNECT BY* pueden ser vistas en Gennick [27].

En DB2 se puede usar el operador *WITH* proveniente del estándar SQL-99 así:

```
WITH temptab(carnet) AS
( SELECT v.carnet
FROM cliente v
WHERE carnet = 81
UNION ALL
SELECT sub.carnet
FROM cliente sub, temptab super
WHERE sub.recomendado_por = super.carnet)
SELECT COUNT(*)-1 AS total
FROM temptab;
```

Las consultas recursivas pueden surgir en muchos contextos

Los SGBD ofrecen extensiones propietarias para solucionar las consultas recursivas

Se propone el siguiente método: a cada cliente se le asignará una *clave inteligente* llamada *ruta*, la que permitirá de una manera "camuflada" mantener el camino de los ancestros de cada individuo. Las letras del alfabeto servirán para este propósito. Por ejemplo, al cliente que da lugar al nacimiento de todo el árbol (Ana en este caso) se le asigna la ruta con la letra *a*. A sus "hijos", León y Carlos se les asignan respectivamente las cadenas *aa* y *ab*. A los hijos de Carlos: Lisa, María y Pedro; se les asignarán las cadenas *aba*, *abb* y *abc* respectivamente. La lista completa de asignaciones de claves inteligentes se muestra en la Tabla III.

Tabla III. Adición de clave inteligente ruta a la relación Cliente

Adición de clave inteligente ruta a la relación Cliente			
carné	**nombre**	**recomendado por**	**ruta**
13	Ana	NULL	a
23	León	13	aa
81	Carlos	13	ab
12	Héctor	23	aaa
14	Juan	23	aab
5	Lisa	81	aba
6	María	81	abb
33	Pedro	81	abc
9	Luis	14	aaba
17	Dora	6	abba
27	Iván	5	abaa
39	Rosa	27	abaaa

3.1. El aspecto de la inserción

En este punto se presenta un problema: la inserción de los valores en la columna *ruta*. Por supuesto que el usuario final debe ser liberado del mantenimiento de dicha columna. Sería muy incómodo y además susceptible de errores realizar dicha inserción "manualmente".

Para facilitar el proceso de inserción se añadirá a la tabla otra columna. Esta columna se denominará *próximo* y su propósito es guardar para cada cliente el valor correspondiente a la próxima letra que deberá asignársele al siguiente cliente que él recomiende directamente.

Por ejemplo, si Carlos recomienda a un nuevo cliente llamado Tomás con carnet 85, como Carlos ya tiene 3 "hijos": Lisa, María y Pedro; con cadenas *aba*, *abb* y *abc* respectivamente, entonces el próximo hijo de Carlos tendrá como cadena *abd*. Por lo tanto, antes de la inserción de Tomás la tabla Cliente luce como se muestra en la Tabla IV. Luego de la inserción de Tomás, la situación estará así: una nueva fila para Tomás (*85, Tomás, 81, abd, a*) y la fila de Carlos en su columna *próximo* cambia la *d* por la *e*. El resto de la relación Cliente continúa intacta.

Tabla IV. Adición de la columna próximo a la relación Cliente

carné	**nombre**	**recomendado por**	**ruta**	**próximo**
13	Ana	NULL	a	c
23	León	13	aa	c
81	Carlos	13	ab	d
12	Héctor	23	aaa	a
14	Juan	23	aab	b
5	Lisa	81	aba	b
6	María	81	abb	b
33	Pedro	81	abc	a
9	Luis	14	aaba	a
17	Dora	6	abba	a
27	Iván	5	abaa	b
39	Rosa	27	abaaa	a

3.1.1 Reglas de inserción

Se definen dos reglas básicas para toda inserción: (i) todo registro que ingresa tiene su campo *próximo* igual a '*a*', (ii) cuando se va a ingresar un registro que es raíz de una jerarquía, su campo *recomendado_por* debe ser *nulo* (*NULL*) y su campo *ruta* igual a '*a*'. Ejemplo: *INSERT INTO cliente VALUES (13,'Ana', NULL,'a','a');* (iii) para los registros no raíces, su campo *ruta* debe hacerse igual a la ruta de su registro padre (cliente que lo recomienda) concatenado con el valor del campo *próximo* y debe actualizarse el valor del campo *próximo* del registro padre a la siguiente letra. Por lo tanto se requiere una inserción y una actualización. Para el caso de León el proceso es:

```
INSERT INTO cliente VALUES(23,'León',13,'a',
(SELECT ruta||proximo
FROM cliente WHERE
carnet = 13));
UPDATE cliente SET proximo =
CHR(ASCII(proximo)+1) WHERE carnet = 13;
```

3.2. Consultas

Sea la consulta "obtener los clientes que han sido recomendados por clientes que han sido recomendados por el cliente con carnet 81 (es decir los "nietos" del cliente con carnet 81)". Sin utilizar las columnas *ruta* y *próximo* la respuesta clásica a esta consulta implica una reunión de la tabla consigo misma así:

```
SELECT v2.carnet, v2.nombre
FROM cliente v1, cliente v2
WHERE v1.recomendado_por = 81 AND
v1.carnet = v2.recomendado_por;
```

El *EXPLAIN PLAN* en Oracle para esta consulta se puede observar en la Tabla V. Como puede verse hay una reunión (*join*) que se ha realizado mediante el método *Nested Loops*. Antes de realizar la reunión se realiza un *TABLE ACCESS FULL CLIENTE* a la tabla Cliente.

Con el método propuesto la consulta se puede expresar así:

SELECT carnet,nombre
FROM cliente
WHERE ruta LIKE 'ab__';

El *EXPLAIN PLAN* correspondiente a esta consulta se muestra en la Tabla VI.

Tabla V. EXPLAIN PLAN para consulta de la relación Cliente con auto-reunión

SELECT STATEMENT
NESTED LOOPS
TABLE ACCESS FULL CLIENTE
TABLE ACCESS BY INDEX ROWID CLIENTE
INDEX UNIQUE SCAN SYS_C0011084
SELECT STATEMENT

Tabla VI.EXPLAIN PLAN para consulta de la relación Cliente sin auto-reunión

SELECT STATEMENT
TABLE ACCESS FULL CLIENTE

En este caso no hay reunión y sólo se realiza el paso correspondiente al *TABLE ACCESS FULL CLIENTE*. Para obtener resultados cuantitativos y evaluar el consumo de recursos de ambas consultas se podría trabajar con la herramienta *Tkprof*, sin embargo a partir de los *EXPLAIN PLANs* presentados es fácil deducir que la segunda consulta consume menos recursos ya que se evita la reunión.

A continuación se presentan una serie de consultas típicamente recursivas basadas en la tabla Cliente y que se resuelven haciendo uso del método expuesto mediante el uso de las columnas *ruta* y *próximo*. Se presenta el enunciado de la consulta, el enunciado equivalente usando terminología de árboles [15] y la solución correspondiente.

Consulta 1: imprimir todos los clientes (directos e indirectos) que ha recomendado un cliente dado, es decir, todos los descendientes de un nodo dado. Ejemplo: para el Cliente con carnet 81.

SELECT carnet,nombre
FROM cliente
WHERE ruta LIKE 'ab_%';

Consulta 2: imprimir todos los clientes directos que ha recomendado un cliente, es decir, los descendientes directos de un nodo dado. Ejemplo: para el Cliente con carnet 81.

SELECT carnet,nombre
FROM cliente
WHERE ruta LIKE 'ab_';

Consulta 3: imprimir todos los clientes indirectos que ha afiliado un cliente dado, es decir, los descendientes indirectos de un nodo dado. Ejemplo: para el cliente con carnet 81.

SELECT carnet,nombre
FROM cliente
WHERE ruta LIKE 'ab_%' AND
ruta NOT LIKE 'ab_';

Mediante el uso de los comodines '_' y '%' se pueden obtener sólo los nietos o los bisnietos etc.; de un nodo. En el método propuesto por Moreau [25] y Ben-Gan [28], para obtener los nietos de un nodo se requiere una subconsulta o una reunión, los cuales son innecesarios en el método aquí propuesto.

Consulta 4: ¿cuál es la longitud de la cadena de recomendados más larga?, es decir, la altura del árbol.

SELECT MAX(LENGTH(ruta))
FROM cliente;

Consulta 5: ¿cuáles clientes no han recomendado a otros clientes?, es decir, las hojas del árbol.

SELECT carnet, nombre
FROM cliente v1
WHERE proximo = 'a';

Consulta 6: ¿cúal es el número máximo de clientes recomendados directamente por un mismo cliente?, es decir, el grado del árbol.

SELECT MAX(ASCII(proximo)–97)
FROM cliente;

Acá se está sacando provecho de la organización alfabética. Se obtiene el valor ASCII de la columna *próximo* y se le resta el valor 97 (código ASCII de la '*a*'). La mayor diferencia que se obtenga corresponderá al valor de la columna *próximo* que contenga la letra más alta, por lo tanto el cliente que ha recomendado a más clientes.

Consulta 7: dados los carnets de 2 clientes, determinar si han sido recomendados directamente por el mismo cliente, o sea, decir si 2 nodos son hermanos. Por ejemplo para los clientes con carnets 5 y 33.

SELECT 'Si'
FROM cliente a, cliente b
WHERE a.carnet=5 AND b.carnet=33 AND
SUBSTR(a.ruta,1,LENGTH(a.ruta)-1)=
SUBSTR(b.ruta,1,LENGTH(b.ruta)-1);

El método propuesto evita en lo posible la realización de autoreuniones

Consulta 8: imprimir la ruta de recomendación que conduce hasta un cliente determinado,

El método presenta aspectos que merecen ser mejorados

es decir, los ancestros de un nodo. Ejemplo: ruta de recomendación hasta Iván (carnet 27).

```
SELECT carnet, nombre
FROM cliente v1
WHERE (SELECT ruta FROM
cliente WHERE carnet = 27)
LIKE v1.ruta || '_%';
```

Consulta 9: dados los carnets de 2 clientes determinar los clientes comunes en la cadena de recomendación de ambos, es decir, los ancestros comunes a un par de nodos. Ejemplo: para Rosa (carnet 39) y Pedro (carnet 33).

```
SELECT carnet,nombre, ruta AS comun
FROM cliente v1
WHERE ( SELECT ruta
FROM cliente WHERE carnet = 39) LIKE
 v1.ruta || '_%' AND ( SELECT ruta FROM
cliente WHERE carnet = 33) LIKE v1.ruta || '_%';
```

Consulta 10: imprimir en orden, de arriba abajo, la jerarquía de recomendación, es decir, imprimir un árbol por niveles desde la raíz.

```
SELECT carnet, nombre,
LENGTH(ruta) as nivel
FROM cliente
ORDER BY 3;
```

Nótese que la gran mayoría de consultas ejemplificadas no requieren reuniones, lo cual incide en su tiempo de respuesta.

4. DESVENTAJAS Y MEJORAS AL MÉTODO

El método presenta algunos aspectos que pueden ser mejorados, los cuales se describen a continuación.

4.1. Almacenamiento

Dado el espacio requerido por la columna *ruta* se deben proponer técnicas para comprimirla en caso de que la cantidad de datos a manejar sea enorme. La columna *próximo* también requiere espacio, aunque emplea un único carácter.

4.2. Análisis del modelo

Desde el punto de vista de *análisis del modelo*, la presencia de las columnas *ruta* y *próximo* no es "natural". Éste es un factor menor pero debe ser advertido en una etapa temprana y debe ser bien documentado su propósito.

4.3. Representación del árbol

Una de las limitaciones del método planteado, es que un nodo sólo puede tener como máximo 26 hijos (las letras del alfabeto). En el caso de estudio esto equivale a decir que un cliente sólo puede recomendar *directamente* máximo a 26 clientes. Una forma de eliminar esta restricción es que en vez de manejar los hijos de un nodo como *aa, ab, ac, …, az* se puede adicionar un entero positivo antes de la segunda letra así: *a1a, a1b, a1c, … , a1z, a2a, a2b, a2c, …, a2z, a3a, a3b, …, a3z,…, a10a, a10b, a10z, a11a, etc*. De esta manera un nodo puede tener potencialmente infinitos hijos. Por supuesto la cantidad de almacenamiento aumenta pero permite seguir utilizando el método propuesto.

Otro aspecto tiene que ver con la altura del árbol, ya que estará limitada por el máximo número de caracteres que se pueda representar en una cadena (columna *ruta*). Este problema es más fácil de solucionar, por ejemplo se podría usar una segunda cadena de caracteres (*ruta2*) y concatenarlas en las consultas cuando sea necesario. De todas formas los límites actuales para cadenas de caracteres son altos en los SGBDs (2000, 4000 y más).

Con respecto al número de jerarquías (árboles) que se pueden representar en una tabla, este aspecto también se puede solucionar. Supóngase los datos mostrados en la Figura 3.

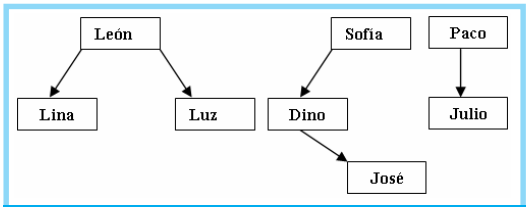

Figura 3. Diversas jerarquías de recomendación.

En este caso se tienen 3 jerarquías, "lideradas" por León, Sofía y Paco. Para solucionar el problema de representación en la columna *ruta*, a cada jerarquía se le asigna un número que se concatena a la columna *ruta*, tal y como lo muestra la Tabla VII.

Tabla VII. Diversas jerarquías de recomendación - Cliente

carné	nombre	recomendado por	ruta	próximo
1	León	NULL	0-a	c
2	Lina	1	0-aa	a
4	Luz	1	0-ab	a
8	Sofía	NULL	1-a	c
9	Dino	8	1-aa	a
3	José	9	1-ab	a
5	Paco	NULL	2-a	b
6	Julio	5	2-aa	a

Con este sistema se pueden manejar teóricamente infinitos árboles. Por supuesto, mientras más árboles existan se requerirán números con mayor cantidad de dígitos al inicio de la *ruta*,

pero en una situación real, considérese 1 millón de árboles, habrá algunos árboles cuya ruta inicia por 6 dígitos, lo cual es manejable. Obviamente el costo de almacenamiento y de mantenimiento se incrementa, pero las consultas propuestas pueden seguir siendo utilizadas (algunas de ellas requerirán algunos "ajustes", por ejemplo la consulta 6 de la sección 3.2, entre otras).

5. CONCLUSIONES Y TRABAJOS FUTUROS

Se ha presentado un método alternativo para el manejo de consultas recursivas en SQL. Con él se pueden resolver con relativa facilidad y de manera eficiente, problemas que de lo contrario requerirían acudir a lenguaje procedimental, a operadores especializados o a consultas basadas en autoreuniones. Se planea realizar un estudio de rendimiento entre el uso de *CONNECT BY* y el método propuesto.

Si el método propuesto ofrece buen rendimiento frente al uso de *CONNECT BY*, se podría concebir un mecanismo entre las dos notaciones que trasforme consultas realizadas con *CONNECT BY* a consultas realizadas con el método propuesto.

Deben también tratarse los casos de eliminación y actualización (¿qué hacer con los clientes que fueron recomendados por un cliente que ya no estará con la compañía?). El método propuesto podría ser utilizado incluso en otros dominios donde la utilización de árboles sea necesaria, por ejemplo en las jerarquías de representación de XML (representación DOM) y en árboles de decisión.

Otras variantes y mejoras, como las expuestas en la Sección 4 pueden igualmente ser introducidas y formas más eficientes para representar la columna *ruta* podrían ser diseñadas para ahorrar almacenamiento (técnicas de compresión). Finalmente se tiene planeado realizar una comparación de rendimiento entre diversos métodos que permitan definir consultas recursivas como los expuestos en la Sección 1.

REFERENCIAS BIBLIOGRÁFICAS

[1] Jim Melton, *Understanding SQL's Stored Procedures*, Primera Edición, Morgan Kaufmann, San Francisco, California, 1988, pp. 10-30.

[2] Oracle Corporation, *Oracle9i SQL Reference Release 2 (9.2)*, Primera Edición, Oracle, Redwood City, California, 2002, pp. 361-366.

[3] Torsten Steinbach, *Migrating Recursive SQL from Oracle to DB2 UDB*, Julio de 2003, disponible en http://www.ibm.com/developerworks/db2/library/techarticle/0307steinbach/0307steinbach.html

[4] Peter Gulutzan y Trudy Pelzer, *SQL-99 Complete Really*, Primera Edición, R & B Books, Lawrence, Kansas, 1999, pp. 627-630.

[5] Jeffrey Ullman y Jennifer Widom, *A First Course in Database Systems*, Tercera Edición, Prentice Hall, Stanford, Kentucky, 2001, pp. 492-498.

[6] Leonid Libkin, "Expressive Power of SQL", *Theoretical Computer Science*, Volumen 3, Número 296, 2003, pp. 379-404.

[7] Chris Date, "A Note on the Parts Explosion Problem", *Relational Databases: Selected Writings*, Addison-Wesley, 1986.

[8] Rakesh Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *IEEE Transactions on Software Engineering (TSE)*, Volumen 14, Número 7, 1988, pp. 879-885.

[9] Rafiul Ahad y Bing Yao, "RQL: A Recursive Query Language", *IEEE Transactions on Knowledge and Data Engineering*, Volumen 5, Número 3, 1993, pp. 451-461.

[10] Ming-Chien Shan y Marie-Anne Neimat, "Optimization of Relational Algebra Expressions Containing Recursion Operators", *ACM Annual Computer Science Conference*, 1999, pp. 332-341.

[11] Volker Linnemann, "Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach", *ICDE*, 1987, pp. 591-598.

[12] John Tillquist y Feng-Yang Kuo, "An Approach to the Recursive Retrieval Problem in the Relational Database", *Communications of the ACM (CACM)*, Volumen 32, Número 2, 1989, pp. 239-245.

[13] Michael Stonebraker, Eugene Wong, Peter Kreps y Gerald Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems (TODS)*, Volumen 1, Número 3, 1976, pp. 189-222.

[14] H. V. Jagadish, Rakesh Agrawal y Linda Ness, "A Study of Transitive Closure as a Recursion Mechanism", *SIGMOD Conference*, 1987, pp. 331-344.

[15] Ellis Horowitz, Sartaj Sahni y Dinesh Mehta, *Fundamentals of Data Structures in C++*, Segunda Edición, W. H Freeman, Miami, Florida, 2006, pp. 350-364.

[16] David Maier y David Scott Warren, *Computing with Logic: Logic Programming with Prolog*, Primera Edición, Benjamin/Cummings, Miami, Florida, 1988, pp. 10-33.

[17] Kemal Koymen y Qujun Cai, "SQL*: a Recursive SQL", *Information Systems*, Volumen 18, Número 2, 1993, pp. 121-128.

[18] Arnon Rosenthal, Sandra Heiler, Umeshwar Dayal y Frank Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", *SIGMOD Conference*, 1986, pp. 166-176.

[19] G. James y W. Stoeller, "Operations on Tree-structured Tables", *X3H2-26-15 Standards Communication*, 1982.

[20] Joachim Biskup, Uwe Räsch y Holger Stiefeling, "An Extension of SQL for Querying Graph Relations", *Computer Languages*, Volumen 15, Número 2, 1990, pp. 65-82.

[21] Isabel F. Cruz, Alberto O. Mendelzon y Peter T. Wood, "A Graphical Query Language Supporting Recursion", *SIGMOD Conference*, 1987, pp. 323-330.

[22] Carlos Ordonez, "Optimizing Recursive Queries in SQL", *SIGMOD Conference*, 2005, pp. 834-839.

[23] Joe Celko, *SQL for Smarties*, Tercera Edición, Morgan-Kaufmann, San Francisco, California, 2005, pp. 623-640.

[24] Joe Celko, *Trees & Hierarchies in SQL*, Primera Edición, Morgan-Kaufmann, San Francisco, California, 2003, pp. 45-99.

[25] Tom Moreau y Itzik Ben-Gan, *Advanced Transact-SQL for SQL Server 2000*, Primera Edición, APress, Miami, Florida, 2000, pp. 579-628.

[26] Chris Date, *Introducción a los Sistemas de Bases de Datos*, Séptima Edición, Prentice Hall, Welmintong, Delaware, 2001, pp. 107-108.

[27] Jonathan Gennick, *New Connect by Features in Oracle Database 10g*, Septiembre de 2003, disponible en http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/gennick_connectby_10g.html

[28] Itzik Ben-Gan, *Maintaining Hierarchies*, Julio de 2000, disponible en http://www.sqlmag.com/Articles/Index.cfm?ArticleID=8826&pg=1

Francisco Javier Moreno Arboleda

Ingeniero de Sistemas, U. de Antioquia. Especialista en Gestión y Sistemas de Bases de Datos de la U. de Antioquia. Magíster en Ingeniería de Sistemas en la Universidad Nacional de Colombia sede Medellín, Colombia. Actualmente es docente en el área de Bases de Datos en la Universidad Nacional de Colombia, sede Medellín y adelanta su Doctorado en Ingeniería - Sistemas e Informática en la misma universidad. **fjmoreno@unal.edu.co**

Jaime Alberto Guzmán Luna

Ingeniero Civil y Magíster en Ingeniería de Sistemas de la U. Nacional de Colombia, sede Medellín. Especialista en Comunicación Educativa de la Universidad de Pamplona. Actualmente es docente en el área de Objetos en la U. Nacional de Colombia, sede Medellín y adelanta su Doctorado en Ingeniería - Sistemas e Informática en la misma universidad. **jaguzman@un.edu.co**

El rendimiento y expresividad del método debe compararse con otras propuestas