Universidad Latinoamericana de Ciencia y Tecnología

Facultad de Ingeniería

Escuela de Ingeniería Informática

TRABAJO FINAL PARA OPTAR POR EL GRADO ACADÉMICO DE LICENCIATURA EN SISTEMAS INFORMÁTICOS CON ÉNFASIS EN DESARROLLO DE SOFTWARE

Tema: Optimización del desempeño en sentencias SQL

Sustentante: Marco Baltodano García

Cédula: 1-1160-0661

Tutor: Lic. Miguel Pérez Montero

San José, Costa Rica
Abril, 2009
I Cuatrimestre, 2009

1 Tabla de Contenido

2 Introducción

Es muy probable de que en algún momento dentro de la Carrera de Ingeniería en Sistemas Informáticos, alguna de las tareas involucre utilizar un sistema de administración base de datos (*Database Management System* o por sus siglas en inglés, DBMS referido así en adelante) como repositorio de datos, y es muy probable de que se encuentre con alguno de los siguientes DBMS que dominan el mercado:

Tabla 1. Distribución del mercado y crecimiento en todos los sistemas operativos, 2006 y 2007 (en millones de dólares)

	2006	2007	Distribución (%) 2006	Distribución (%) 2007	Crecimiento (%) 2006-2007
Oracle	7 212,1	8 287,9	47,9	48,6	14,9
IBM	3 206,4	3 526,0	21,3	20,7	10,0
Microsoft	2 654,4	3 092,4	17,6	18,1	16,5
Otros	1 993,2	2 163,3	13,2	12,7	8,5
Total	**15 066,1**	**17 069,6**	**100**	**100**	**13,3**

Fuente: (Graham, 2008)

También, es altamente presumible de que también se encuentre con otros DBMS que son muy populares, tales como Teradata, Sybase, Progress y MySQL, entre otros. El propósito del desarrollo de esta investigación, es brindar una guía de optimización que permita ser aplicada en los diferentes DBMS, más específicamente, en los sistemas de administración de bases de datos relacionales (*Relational Database Management System* o por sus siglas en inglés, RDBMS referido así en adelante).

En una aplicación comercial que utilice un RDBMS como repositorio de datos es muy frecuente que se examinen millones de registros en la base de datos sólamente para retornar unos pocos registros que

cumplen con las condiciones originales de búsqueda que se enviaron desde la aplicación, y es esta ineficiencia la que puede hacer que el procesamiento del sistema se sobrecargue y su desempeño decaiga. Por esto, es imperativo establecer que su aplicación comercial se adecue a sus volúmenes de procesamiento de datos. Y la manera más segura de hacer esto, es mejorar el desempeño de las sentencias SQL, ya que los cambios que esto implica tienen efectos casi imperceptibles en el código de una aplicación estable.

A su vez, los problemas de desempeño se traducen directamente en pérdida de la productividad, ya que los usuarios finales perderán tiempo esperando por una respuesta del sistema. Y esto tiene varios efectos, ya que los usuarios se vuelven poco productivos y necesitarán de mayor tiempo para poder finalizar con sus labores, añadiendo un cierto grado de desmotivación al personal.

La relación de costo-beneficio es tan positiva que todas las aplicaciones comerciales basadas en RDBMS deberían tener sus sentencias SQL de alto volumen optimizadas.

Sin embargo, será necesario un conocimiento básico en programación y el estándar en sintaxis de sentencias SQL. Adicionalmente, se confirmarán las observaciones que se indiquen, con pruebas reales y datos concretos en los diferentes RDBMS.

3 Metodología

Para evitar el problema de confinar las recomendaciones a un vendedor en específico, se utilizarán ejemplos con sentencias SQL, siguiendo el estándar ANSI/ISO SQL:1999. Algunos ejemplos de herramientas cliente serán indicados con lenguaje JAVA, utilizando JDBC.

4 Discusión

4.1 Optimización Basada en Sintaxis

A continuación se enfocará el nivel más bajo de optimización al que suele recurrirse: la optimización basada en sintaxis.

La sintaxis de un lenguaje de programación define la forma correcta de escribir las sentencias y los datos de cualquier programa (Berlanga Llavori, Inesta Quereda, García Sevilla, Gracia Luengo, & Barber Miravalles, 2000). Para optimizar basándose en sintaxis, debe asumirse que todos los factores no sintácticos (índices, tamaños de tablas y almacenamiento) son desconocidos e irrelevantes. La ventaja de este tipo de optimización, es que puede realizarse en el cliente. Es importante indicar que este tipo de optimización se centra en la condición de búsqueda de la sentencia SQL, ya que esta sintaxis particular ofrece mayores posibilidades de optimización.

Sin embargo, como primera regla de oro, debe entenderse el código antes de intentar optimizar una sentencia SQL. Por ejemplo, en programas viejos de SQL, podría presentarse la siguiente sentencia:

```
SELECT * FROM Tabla
 WHERE 0 = 1
   AND columna = 'Sentencia documentada'
```

Esto sucede, porque hace mucho tiempo atrás, los comentarios en el lenguaje SQL indicados con /* y */ no eran permitidos, por lo que muchos desarrolladores optaban con documentar sus sentencias con condiciones que las hacían completamente falsas para evitar a que se ejecutaran.

4.1.1 Optimización General

Para iniciar con el tema de optimización es necesario iniciar desde lo sencillo y general, hasta lo complejo y específico.

A continuación se examinará algunas ideas generales que deben tomarse en cuenta cuando se escriben condiciones simples de búsqueda:

4.1.1.1 Sargable

Las mejores condiciones de búsqueda son aquéllas que trabajan sobre pocos registros en comparaciones sencillas, y toman menos tiempo en realizarse. Como una regla general, debe buscarse que la condición de búsqueda esté construida al lado izquierdo por una columna simple, y al lado derecho por un valor que se pueda obtener fácilmente. Por ejemplo:

```
SELECT * FROM Tabla

 WHERE columna1 = 123456
```

Así que la forma general de una condición de búsqueda ideal sería la siguiente:

<columna> <operador de comparación> <valor literal>

Lo anterior se conoce como expresión sargable, que viene de la acotación en inglés "*search argument-able*" (o búsqueda argumental, traducido al español), lo que significa que el predicado especifica un argumento de búsqueda que puede hacer uso efectivo de un índice (Carter, 2004). En otras palabras, una expresión sargable es buena, y una no-sargable es perjudicial para la optimización del desempeño de una sentencia SQL.

4.1.1.2 Ley de la Transitividad por Transferencia

Utilizar la ley de la transitividad por transferencia, que establece que si un objeto X es mayor (o menor) que el objeto Y, al mismo tiempo que éste es mayor (o menor) que otro objeto Z,

entonces el primer objeto X es mayor (o menor) que el tercero Z (Gortari, 1988). O bien, visto desde el punto de vista de sintaxis en SQL:

```
IF (X <operador> Y) IS TRUE AND (Y <operador> Z) IS TRUE
THEN
(X <operador> Z) IS TRUE
```

Donde el operador de comparación es: =, >, >=, <, <=. Pero no es: <> o LIKE.

Con la ley anterior, puede realizarse una transformación de sentencias SQL (al referirse a transformación, significa que se busca reescribir la sentencia SQL para producir el mismo resultado, pero con una sintaxis diferente):

Expresión 1:

```
WHERE columna1 < columna2
  AND columna2 = columna3
  AND columna1 = 8
```

Expresión 2:

```
WHERE 8 < columna2
AND columna2 = columna3
AND columna1 = 8
```

Otro ejemplo de esta idea para una sentencia que se ejecute muy lentamente:

Expresión 1:

```
SELECT * FROM Tabla
WHERE columna1 = 8
   AND NOT (columna3 = 9 OR columna1 = columna2)
```

Aplicando la transformación, puede tenerse una sentencia como la siguiente:

Expresión 2:

```
SELECT * FROM Tabla
WHERE columna1 = 8
  AND columna3 <> 9
  AND columna2 <> 8
```

La sentencia anterior transformada se ejecuta dos veces más rápida que la sentencia original.

4.1.1.3 Eliminación de Constantes Propagadas

Cualquier persona que haya utilizado un lenguaje de programación entenderá que la siguiente expresión X=8+8-16 es convertida a X=0, por propagación de constantes, en tiempo de ejecución. Así que es sorprendente saber que muchos DBMS no convierten los siguientes cinco ejemplos, de cambios evidentes:

```
…WHERE columna1 + 0

…WHERE 8 + 0.0

…WHERE columna1 IN (8, 10, 10)

…CAST (8 AS INTEGER)
```

Si se encuentra con estas expresiones en códigos SQL de programas antiguos, la recomendación es ignorarlos. Estas sentencias se han escrito por razones, tales como forzar al DBMS a ignorar indexación, cambiar el tipo de dato del resultado o especificar una diferencia entre tipos de datos SMALLINT (enteros pequeños) e INTEGER (enteros). Por ello, es bueno recordar la primera regla de oro: primero debe entenderse el código antes de optimizarlo.

Sin embargo, sí se recomienda optimizar sentencias SQL como la siguiente condición de búsqueda:

```
…WHERE columna1 – 5 = 8
```

a:

```
…WHERE columna1 = 13.
```

Con la anotación de que la sentencia anterior se ejecutará dos veces más rápida en la mayoría de los DBMS actuales.

Así mismo, cuando existen operaciones de aritmética involucradas, es importante transformar expresiones como la siguiente:

…WHERE columna1 – 8 = -columna2

Transformada a:

…WHERE columna1 = -columna2 + 8.

Es importante recalcar que expresiones que involucren operaciones aritméticas, se ejecutarán más rápido si todos sus operandos son idealmente de tipo INTEGER (entero), en lugar de SMALLINT (enteros pequeños), DECIMAL, o FLOAT (decimal de coma flotante). Así que la expresión:

```
…WHERE columna_flotante * columna_decimal;
```

Se ejecutará más lenta que la siguiente expresión:

```
…WHERE columna_entera * columna_entera;
```

Por lo tanto, el diseño de la base de datos también es un factor importante por tomar en cuenta para lograr un óptimo desempeño de las sentencias SQL.

4.1.2 Optimización Específica

Luego de haber examinado algunas ideas generales de cómo optimizar las condiciones de búsqueda, se analizará la optimización de ciertos operadores de SQL:

4.1.2.1 AND

La mayoría de los DBMS evaluarán una serie de expresiones AND de izquierda a derecha, así que si la primera expresión es falsa, el DBMS no continuará evaluando las siguientes expresiones que le preceden. Tomando en cuenta este comportamiento, puede optimizarse la condición de búsqueda de la sentencia SQL, colocando la expresión AND que tiene menor probabilidad de ser verdadera al inicio. Por ejemplo, asuma una tabla de empleados, en donde se almacena su salario y edad entre otros atributos de éste, considere la siguiente expresión:

```
… WHERE salario >= 500000 AND edad > 80
```

La misma puede transformarse a:

```
… WHERE edad > 60 AND salario >= 500000
```

Donde se conoce que dentro de los registros consultados, la mayoría de las edades de los empleados oscilan entre 20 y 50 años, es muy poco probable de que se encuentren registros de empleados con edades mayores a 60 años.

4.1.2.2 OR

Al contrario de las expresiones AND, en las expresiones OR deben indicarse las expresiones con mayor probabilidad de ser verdadera al inicio, dado que si la primera expresión resulta falsa, se continuará evaluando la segunda expresión y así sucesivamente. Retomando el ejemplo anterior, y asumiendo que en la tabla de empleados, la mayoría de ellos perciben un salario mayor o igual a 500 000 colones, entonces, puede transformarse la siguiente condición de búsqueda:

```
… WHERE edad > 80 OR salario >= 500000 AND
```

De la siguiente manera:

```
… WHERE salario >= 500000 OR edad > 60
```

Dado que la mayoría de los empleados percibe un salario mayor a 500 000 colones, es muy probable que la primera expresión OR evaluada sea verdadera, con lo que el DBMS no continuará evaluando las siguientes expresiones OR.

4.1.2.3 IN

El operador IN en SQL ofrece la habilidad de permitir múltiples condiciones en una sentencia SQL (Dixit, 2008). Por ejemplo, la siguiente sentencia permite obtener un listado de pagos que se encuentran en estado ‘cancelado’ o ‘inválido’:

```
SELECT numero_pago, fecha_pago

FROM PAGOS

WHERE estado IN (‘cancelado’, ‘invalido’)
```

La lista de los valores entre paréntesis redondos es llamada *enlista*.

En la mayoría de los DBMS, se acostumbra a observar que las siguientes expresiones son idénticas:

Expresión 1:

```
…WHERE nombre_provincia = 'heredia'

OR nombre_provincia = 'alajuela'
```

Expresión 2:

```
…WHERE nombre_provincia IN ('heredia', 'alajuela').
```

Sin embargo, en algunos DBMS la segunda expresión se ejecuta más rápido y, en el resto, se transforma automáticamente a la primera (que utiliza OR), por lo que resulta una muy buena idea utilizar la expresión IN, en lugar de OR.

Por otro lado, en un rango denso de valores enlistados, es recomendable analizar cuál será la mejor manera de definir la condición de búsqueda. Por ejemplo, la siguiente expresión:

```
…WHERE anno_contratacion IN (2003, 2004, 2005, 2006, 2008, 2009)
```

Contrasta enormemente en desempeño si se reescribe de la siguiente manera:

```
…WHERE anno_contratacion BETWEEN 2003 AND 2009

AND anno_contratacion <> 2007
```

Ya que la segunda expresión, se ejecutará más rápido que la primera, al tener el DBMS que analizar menos comparaciones.

4.1.2.4 UNION

La cláusula SQL UNION se utiliza para unir o concatenar el resultado de múltiples sentencias SELECT (Dam, 2004), como por ejemplo:

```
SELECT código_producto, descripción_producto FROM PRODUCTOS WHERE
código_suplidor = 1

UNION

SELECT código_producto, descripción_producto FROM PRODUCTOS WHERE
código_suplidor = 2
```

La cláusula UNION procesa el grupo de resultados de las sentencias SELECT, removiendo duplicados del grupo de resultados final (Dam, 2004), con lo cual se aumenta el tiempo de procesamiento para obtener los registros resultantes. Si se sabe que ambos grupos de resultados son excluyentes, y no habrá dos o más registros duplicados; o, en su defecto, que no sea necesario filtrar registros duplicados en el grupo de resultados final, se recomienda reemplazar la cláusula UNION por UNION ALL, con lo que se traducirá en un aumento del desempeño considerable.

4.1.2.5 Optimización de Recuento de Registros

Usualmente, COUNT(*) es la técnica apropiada para contar el número de registros en una tabla. En algunos casos, pueden encontrarse situaciones cuando se utiliza la sentencia COUNT(<columna>) para contar el número de registros de una columna en particular (Dam, 2004). Cuando se utiliza la versión de COUNT(*), se le permite al optimizador interno del DBMS seleccionar el mejor índice disponible de la tabla; por lo tanto, se asegura el mejor desempeño si los índices son modificados posteriormente.

Existen algunos que aseguran que la utilización de COUNT(1) resulta ser más eficiente en términos de desempeño que el utilizar COUNT(*). Sin embargo, ha logrado demostrarse que los DBMS actuales tratan internamente las dos expresiones de manera similar, por lo que no existe ninguna diferencia real de desempeño entre las dos opciones (AskTom, 2001).

4.2 Consideraciones y Opciones de Ordenamiento

A continuación se examinan algunas consideraciones importantes que deben tomarse en cuenta cuando se definen las opciones de ordenamiento de la sentencia SQL.

4.2.1.1 INTEGER vs. CHAR

Columnas de tipo INTEGER (Números enteros) resultan más rápido de comparar que si se utilizan columnas de tipo CHAR (Caracteres), DATE (Fecha) y comparaciones con valores NULL (Nulos). Por lo tanto, la búsqueda y ordenamiento sobre columnas INTEGER resultarán más eficientes que si se ordenaran columnas CHAR. Dado que los índices no almacenan referencias a valores NULL, las condiciones NULL involucran procesamiento adicional, resultando así en el operando condicional más lento de todos (Rob, Crockett, Crockett Rob, & Coronel, 2008). Consiguientemente, la siguiente expresión:

```
SELECT columna_entera FROM Tabla_A ORDER BY columna_entera
```

Se ejecutará más rápido que esta otra expresión:

```
SELECT columna_caracter FROM Tabla_B ORDER BY columna_caracter.
```

4.2.1.2 Factores que Afectan la Velocidad de Ordenamiento

En orden de importancia, se enumeran los factores que afectan la velocidad de ordenamiento de una sentencia SQL:

1) Número de registros seleccionados.

2) Cantidad de columnas en la cláusula ORDER BY.

3) Longitud de las columnas en la cláusula ORDER BY.

El incremento en la cantidad de registros seleccionados tiene un efecto geométrico. Si se incrementa el número de registros por un factor de diez, la tarea de ordenamiento toma cerca de 20 veces más tiempo en completarse (Gulutzan & Pelzer, 2002).

Sorprendentemente, un incremento en la cantidad de columnas tiene un efecto aún más drástico que el incremento en la longitud de cada una de las columnas dentro de la cláusula ORDER BY.

Así que, en resumen, para mejorar el desempeño de la tarea de ordenamiento en una sentencia SQL, debe tomarse una acción drástica en disminuir la cantidad de registros seleccionados. Para hacerlo, debe irse seleccionando solo partes de la tabla. Debe tomarse una acción severa en reducir la cantidad de columnas en la cláusula ORDER BY (puede concatenarse el contenido de dos columnas en una sola, en lugar de especificarlas por separado). Asimismo, debe tomarse una acción moderada al reducir la longitud de las columnas dentro de la cláusula ORDER BY, utilizando funciones como SUBSTRING, por ejemplo. También, es ligeramente útil si los valores ya están previamente ordenados y son únicos en los primeros caracteres de cada columna (Gulutzan & Pelzer, 2002).

4.3 Uniones

Se realiza una unión o JOIN cuando se recuperan datos de más de una tabla a la vez. Para combinar varias tablas se enumeran en la cláusula FROM de la sentencia SELECT. El DBMS creará el producto cartesiano de cada tabla en la cláusula FROM. Sin embargo, para obtener el resultado correcto, se tienen que seleccionar sólo las filas en las que los valores de los atributos comunes concuerdan (Rob, Crockett, Crockett Rob, & Coronel, 2008).

Una de las ventajas más importantes que ofrecen los DBMS actuales, y una de las bases de la teoría relacional, es el hecho de definir llaves foráneas en tablas hijas que contienen campos que referencian valores a campos de una tabla padre. Se menciona como una ventaja, por que permite delimitar los valores posibles que se almacenarán en la tabla hija, y que deben existir en la tabla padre; y como mayor ventaja, dicha relación está almacenada en el diccionario de datos del DBMS. Dicho diccionario de datos puede ser examinado por un usuario que tenga acceso suficiente a consultarlo, y del que muchas herramientas de diseño gráfico de consultas de bases de datos hacen uso, para facilitar el desarrollo de cláusulas SQL sin ningún esfuerzo extra del diseñador. En otras palabras, las herramientas automáticamente crearán la cláusula de unión de una tabla hija contra una tabla padre y evitarán que el desarrollador tenga que definirlas explícitamente dentro de la consulta. Además, la implementación de llaves foráneas asegura la integridad y la veracidad de los datos almacenados. Por ello, el uso de las llaves foráneas en una base de datos transaccional, es indispensable, tanto así como el uso de llaves primarias dentro de una tabla.

La mayoría de los DBMS comerciales que se utilizan actualmente en el mercado, utilizan las llaves foráneas para construir índices automáticos sobre los campos referenciados, ya que es muy probable

que se realicen consultas que impliquen la unión de dichos campos de la tabla hija con la tabla padre, con lo cual el desempeño de las consultas se verá directamente beneficiado.

Sin embargo, el estilo de definir la cláusula de unión entre dos tablas ha implicado un gran debate desde la publicación del estándar de sintaxis SQL llamado ANSI-SQL-92, que define una manera diferente de establecer las condiciones de unión y condiciones de búsqueda o restricción dentro de la sentencia SQL.

Para unir dos tablas pueden utilizarse dos estilos:

4.3.1 FROM/WHERE (Estilo antiguo)

Un ejemplo de sintaxis, utilizando este estilo, sería el siguiente:

```
SELECT C.ID_Cliente, C.Nombre_Cliente, O.Fecha_Orden, O.Monto_Total
FROM Cliente C, Orden O
WHERE C.ID_Cliente = O.ID_Cliente
AND O.Activa = 'N';
```

La consulta anterior retornaría información acerca del cliente, así como información detallada de las órdenes relacionadas con el cliente, que no estén activas. Nótese que las tablas involucradas se especifican en la cláusula FROM. También, las condiciones de unión (el ID del cliente) y las condiciones de búsqueda o restricción (que la orden no esté activa), se especifican en la cláusula WHERE de la sentencia. Es, por este último aspecto, que el estilo antiguo de unión de tablas ha venido perdiendo aceptación y ha caído en desuso, ya que las condiciones de unión están entremezcladas entre las condiciones de búsqueda o restricción, lo que torna confuso el detectar rápida y claramente cuáles son los criterios que unen las tablas, sobre qué columnas se está realizando la unión de las mismas, y separar éstos, de los criterios de búsqueda o restricción.

Existen otros tipos de uniones o JOIN's como, por ejemplo, la unión externa o OUTER JOIN, que permite seleccionar registros de una tabla A, aun cuando no haya coincidencias en las condiciones de unión contra otra tabla B. Dicho de otra manera, y utilizando la analogía de órdenes y clientes, pueden seleccionarse todos aquellos clientes en la sentencia SQL, aunque cuando éstos no tengan órdenes relacionadas. Utilizando sintaxis de estilo antiguo en bases de datos Oracle para ilustrar el ejemplo anterior, podría tenerse una sentencia SQL como la siguiente:

```
SELECT C.ID_Cliente, C.Nombre_Cliente, O.Fecha_Orden, O.Monto_Total
FROM Cliente C, Orden O
WHERE C.ID_Cliente (+) = O.ID_Cliente
```

Ahora bien, ilustrando el mismo ejemplo, pero esta vez, utilizando sintaxis de estilo antiguo en bases de datos SQL Server, se tendría la siguiente sentencia SQL como resultado:

```
SELECT C.ID_Cliente, C.Nombre_Cliente, O.Fecha_Orden, O.Monto_Total
FROM Cliente C, Orden O
WHERE C.ID_Cliente *= O.ID_Cliente
```

Ambas sentencias funcionan de manera perfecta dentro de cada una de las bases de datos indicadas, pero el problema es que muchas aplicaciones que se desarrollan actualmente, o inclusive reportes analíticos, deberán ser desarrollados de una manera que puedan extraer la información de la misma manera que si consultaran una base de datos Oracle o una SQL Server. Es ahí cuando la sintaxis propietaria de cada DBMS, se vuelve un inconveniente en términos de portabilidad hacia otras plataformas de bases de datos.

Sin embargo, el Instituto Nacional Americano de Estándares, o ANSI, por sus siglas en inglés, ha trabajado, desde 1986, en un estándar de lenguaje de consultas de bases de datos llamado ANSI-

SQL, con sus posteriores revisiones, para permitir la portabilidad de sentencias SQL a través de los diferentes DBMS, de manera que una sentencia en ANSI-SQL, sea sintáctica y semánticamente correcta en los DBMS que soporten dicho estándar. Los DBMS más utilizados comercialmente, sin duda, se han adaptado y aceptan dicho estándar, por lo que resulta una ventaja, en todo sentido, escribir sentencias siguiendo el estándar ANSI-SQL, por lo que, sin duda, es una de las mayores recomendaciones que pueden indicarse dentro de esta investigación.

4.3.2 JOIN (SQL-92 o Estilo nuevo)

En la revisión mayor de 1992 del estándar de ANSI-SQL, se provee de palabras claves para los distintos tipos de uniones (JOIN, CROSS JOIN, NATURAL JOIN, que pueden ser combinados con INNER, RIGHT OUTER, LEFT OUTER y FULL OUTER para montar el espectro completo de las uniones). (Bowman, Emerson, & Darnovsky, 2001).

Todas las uniones de registros de dos o más tablas pueden, generalmente, agruparse en dos grandes grupos: INNER JOIN's y OUTTER JOIN's (Kriegel & Trukhnov, 2007). Describiendo la funcionalidad de cada una de estas palabras claves de manera gráfica, y siguiendo la misma analogía de clientes y órdenes, se repasarán sus diferentes variantes.

Se utilizarán las siguientes tablas que se ilustran a continuación, para ejemplificar las distintas variantes de las cláusulas JOIN:

Tabla 2: cliente

Id_Cliente *	Nombre_Cliente *	Apellido_Cliente *
1	Manuel	Salgado
2	Jose	Pereira
3	Juan	Obando

Fuente: Creación del investigador.

Tabla 3: órdenes

Numero_Orden *	Fecha_Orden *	Monto_Total *	Id_Cliente
1	3/23/2009 12:00:00 AM	1000.00	1
2	3/24/2009 12:00:00 AM	5000.00	1
3	3/24/2009 12:00:00 AM	75000.00	2
4	3/26/2009 12:00:00 AM	100.00	1
5	3/27/2009 12:00:00 AM	3000.00	{null}

Fuente: Creación del investigador.

4.3.2.1 INNER JOIN

Un INNER JOIN, retornará solamente aquellos registros que tienen valores coincidentes en ambas tablas especificadas en la unión, excluyendo los otros registros.

Un ejemplo de una sentencia SQL haciendo uso del INNER JOIN y utilizando las tablas anteriores, sería el siguiente:

```
SELECT cliente.Id_Cliente, cliente.Nombre_Cliente,
       cliente.Apellido_Cliente,orden.Numero_Orden, orden.Fecha_Orden,
       orden.Monto_Total
  FROM cliente INNER JOIN orden
       ON (cliente.Id_Cliente = orden.Id_Cliente);
```

Se recomienda identificar a qué tabla pertenecen cada una de los campos involucrados en la consulta SQL, para agilizar la ejecución de la sentencia SQL, y evitar posibles conflictos cuando un campo tiene un mismo nombre en las dos tablas involucradas (un ejemplo es el campo Id_Cliente, el mismo existe en ambas tablas con el mismo nombre), lo cual es muy probable.

Nótese cómo también suele indicarse las sentencias siguiendo un orden en la identación de la cláusula SQL, lo cual ayuda mucho a aquellas personas que deban, en algún momento, mantener el

código escrito por otra persona. Se aconseja proporcionar un comentario breve de aquellas sentencias SQL que realicen una cantidad considerable de uniones en distintas tablas, para brindar un resumen de la intención de dicha sentencia. Dichos comentarios, sin duda, serán muy útiles, inclusive para el mismo autor, en un futuro.

La cláusula anterior puede reescribirse de manera que se utilicen "alias", los cuales permiten referirse a la tabla dentro de la sentencia SQL, sin tener que escribir su nombre completo para identificar cada uno de los campos involucrados en la consulta. Para ello, se recomienda utilizar "alias" que sean cortos, y que sean representativos para no dejar la menor duda de qué tabla se está haciendo referencia en la sentencia. Reescribiendo el ejemplo anterior, utilizando alias, se tendrá una sentencia como la siguiente:

```
SELECT c.Id_Cliente, c.Nombre_Cliente, c.Apellido_Cliente,
       o.Numero_Orden, o.Fecha_Orden, o.Monto_Total
  FROM cliente c INNER JOIN orden o ON (c.Id_Cliente = o.Id_Cliente);
```

Al ejecutar la sentencia anterior, utilizando los datos de las tablas de ejemplo, se tendrá un grupo de resultados como el siguiente:

Tabla 4: INNER JOIN utilizando alias

Id_Client...	Nombre_Client...	Apellido_Client...	Numero_Orde...	Fecha_Orden *	Monto_Tot...
1	Manuel	Salgado	1	3/23/2009 12:00:00 AM	1000.00
1	Manuel	Salgado	2	3/24/2009 12:00:00 AM	5000.00
2	Jose	Pereira	3	3/24/2009 12:00:00 AM	75000.00
1	Manuel	Salgado	4	3/26/2009 12:00:00 AM	100.00

Fuente: Creación del investigador.

La palabra clave INNER es opcional, de manera que si se omite dentro de la sentencia anterior, el resultado no se verá afectado.

Adicionalmente, existe otra manera de realizar un INNER JOIN de manera natural y abreviada: puede utilizarse la palabra clave NATURAL JOIN para unir dos tablas que contengan columnas con el mismo nombre entre ambas, por lo que no es necesario especificar las columnas que servirán como la condición de unión entre ambas, omitiendo la cláusula ON. Sin embargo, se recomienda no utilizar dicha sintaxis en productos finales, ya que el nombre de las columnas podría, en algún momento, cambiar, con lo que la unión automática entre ambas tablas podría, eventualmente, perderse; y en un ambiente en producción, dicho cambio sería desastroso. Además, al realizar la unión de manera automática, tomando los nombres de las columnas como condición para unirlas, podría tenerse resultados no esperados. Para efectos de ejemplificar la idea anterior, podrían tenerse dos tablas: EMPLEADO y DEPARTAMENTO; ambas contienen una columna llamada ID, y otra llamada NOMBRE. Si se aplicara una sentencia de unión natural entre ambas, los resultados de la misma sentencia no resultarán ser los esperados:

```
SELECT e.ID, e.NOMBRE, e.ID_DEPARTAMENTO, d.ID, d.NOMBRE
  FROM empleado e NATURAL JOIN departamento d;
```

La sentencia anterior retornará cero registros como resultado, lo que obviamente no es lo que se esperaba.

4.3.2.2 OUTER JOIN

El OUTER JOIN se utiliza para retornar **todos** los registros de una tabla A y los registros correspondientes de una tabla B, si es que los hay (Kriegel & Trukhnov, 2008).

El OUTER JOIN tiene distintas cláusulas que pueden utilizarse para modificar la forma en que se realiza dicha unión externa, pero para efectos de mantener el tema simple, se analizarán sólamente tres de sus diferentes formas sintácticas: LEFT OUTER JOIN, RIGHT OUTER JOIN y FULL OUTER JOIN.

El LEFT OUTER JOIN o unión externa izquierda, se utiliza para retornar **todos** los registros de una tabla A (tabla izquierda) que tengan una relación con los registros de una tabla B (tabla derecha), e inclusive aquellos registros de la tabla A que no tengan ninguna relación con los de la tabla B.

Utilizando la analogía de la tabla cliente y órdenes, podría hacerse uso del LEFT OUTTER JOIN, para obtener un listado de los clientes y sus órdenes, incluyendo aquellos clientes que aún no tengan órdenes relacionadas, con la siguiente sentencia SQL:

```
SELECT *
  FROM cliente c LEFT OUTER JOIN orden o
       ON (c.Id_Cliente = o.Id_Cliente);
```

En la sentencia anterior se considera que la tabla cliente es la tabla izquierda, y la tabla orden sería la tabla derecha, al ocupar ambas, dichas posiciones con respecto a la cláusula LEFT OUTER JOIN. Al ejecutar la sentencia anterior, se tendrá un grupo de resultados como el siguiente:

Tabla 5: LEFT OUTER JOIN

Id_Client...	Nombre_Client...	Apellido_Client...	Numero_Orden	Fecha_Orden	Monto_Total	Id_Cliente1
1	Manuel	Salgado	1	3/23/2009 12:00:00 AM	1000.00	1
1	Manuel	Salgado	2	3/24/2009 12:00:00 AM	5000.00	1
1	Manuel	Salgado	4	3/26/2009 12:00:00 AM	100.00	1
2	Jose	Pereira	3	3/24/2009 12:00:00 AM	75000.00	2
3	Juan	Obando	{null}	{null}	{null}	{null}

Fuente: Creación del investigador.

Nótese que en el grupo de resultados final se incluyen aquellos registros de la tabla cliente que aún no tienen registros relacionados con la tabla orden. Dichos registros contienen valores NULOS, al no tener contraparte en la relación.

El RIGHT OUTER JOIN es otra variante de la unión externa y se utiliza para retornar todos los registros de la tabla B (tabla derecha) que tengan una relación con los registros de una tabla A (tabla izquierda), e inclusive aquellos registros de la tabla B, que no tengan ninguna relación con los de la tabla A.

Utilizando el RIGHT OUTER JOIN podría obtenerse un listado de órdenes y la información relacionada con sus clientes, incluyendo aquellas órdenes que aún no tienen clientes asociados, utilizando la siguiente sentencia SQL:

```
SELECT *
  FROM cliente c RIGHT OUTER JOIN orden o
       ON (c.Id_Cliente = o.Id_Cliente);
```

Al ejecutar la sentencia anterior, se obtendrá un grupo de resultados como el siguiente:

Tabla 6: RIGHT OUTER JOIN

Id_Cliente	Nombre_Cliente	Apellido_Cliente	Numero_Orde...	Fecha_Orden *	Monto_Tot...	Id_Cliente1
1	Manuel	Salgado	1	3/23/2009 12:00:00 AM	1000.00	1
1	Manuel	Salgado	2	3/24/2009 12:00:00 AM	5000.00	1
2	Jose	Pereira	3	3/24/2009 12:00:00 AM	75000.00	2
1	Manuel	Salgado	4	3/26/2009 12:00:00 AM	100.00	1
{null}	{null}	{null}	5	3/27/2009 12:00:00 AM	3000.00	{null}

Fuente: Creación del investigador.

Nótese que aparecen enlistadas todas aquellas órdenes, inclusive las que no tienen clientes asociados, por lo que los valores correspondientes de la tabla cliente, serán NULOS. Además, aparecen enlistados solo aquellos clientes que tienen una relación con la tabla orden.

Si quisiera enlistarse todos aquellos clientes y todas las órdenes que hay en la base de datos, puede hacerse uso de la tercera variante: FULL OUTER JOIN. Un ejemplo de esta sentencia SQL sería:

```
SELECT *
  FROM cliente c FULL OUTER JOIN orden o
       ON (c.Id_Cliente = o.Id_Cliente);
```

Ésta daría un resultado como el siguiente:

Tabla 7: FULL OUTER JOIN

Id_Cliente	Nombre_Cliente	Apellido_Cliente	Numero_Orden	Fecha_Orden	Monto_Total	Id_Cliente1
1	Manuel	Salgado	1	3/23/2009 12:00:00 AM	1000.00	1
1	Manuel	Salgado	2	3/24/2009 12:00:00 AM	5000.00	1
1	Manuel	Salgado	4	3/26/2009 12:00:00 AM	100.00	1
2	Jose	Pereira	3	3/24/2009 12:00:00 AM	75000.00	2
3	Juan	Obando	{null}	{null}	{null}	{null}
{null}	{null}	{null}	5	3/27/2009 12:00:00 AM	3000.00	{null}

Fuente: Creación del investigador.

En la tabla de resultados anterior, se incluyen todos los registros de ambas tablas, tengan o no alguna relación en la unión externa. Aquellos registros de una tabla que no posean relación con la otra tabla, aparecerán con valores NULOS en su contraparte.

Sin embargo, no todos los DBMS soportan las tres variantes del OUTER JOIN. Sí es un hecho de que todos los DBMS soportan el LEFT OUTER JOIN. Así que pueden adaptarse las sentencias SQL para buscar el resultado deseado. Por ejemplo, MySQL en su versión 5.1, no soporta FULL OUTER JOIN. Sin embargo, puede lograrse el mismo resultado, con una sentencia como la siguiente:

```
SELECT *
```

```
  FROM cliente c LEFT OUTER JOIN orden o
       ON (c.Id_Cliente = o.Id_Cliente)
UNION
SELECT *
  FROM cliente c RIGHT OUTER JOIN orden o
       ON (c.Id_Cliente = o.Id_Cliente);
```

En la sentencia anterior, se hace uso del UNION para unir los resultados de las dos subsentencias SQL, excluyendo aquellos registros duplicados resultantes luego de unirlas.

Especialmente, en el caso de los OUTER JOIN, se recomienda continuar utilizando la sintaxis que especifica el estándar SQL-ANSI, ya que, por ejemplo, el DBMS de Microsoft SQL Server ha dejado su vieja sintaxis relegada en las versiones 2005 y 2008. Y aun cuando puede cambiarse el nivel de compatibilidad del DBMS a 80 (SQL Server 2000), para que acepte sintaxis de versiones anteriores, no se recomienda seguir este procedimiento (Kriegel & Trukhnov, 2008), ya que puede haber código T-SQL que haga uso de sentencias que fueron implementadas hasta la versión 2005 como, por ejemplo, el manejo de errores con bloques TRY-CATCH, las cuales perderían su funcionalidad al cambiar dicho nivel de compatibilidad.

4.4 Índices

Una de las mejores maneras para incrementar el desempeño de una sentencia SQL, es crear índices útiles. La idea de la utilización de índices radica en que éstos toman menos operaciones de E/S (Entrada/Salida), y consumen menos recursos del sistema para poder localizar los registros en la condición de búsqueda.

Sin embargo, aunque la implementación de índices conlleva una posible mejora en el desempeño de las sentencias SELECT, tiene efectos colaterales en el resto de las sentencias DML (*Data Manipulation Language*, o lenguaje de manipulación de datos), tales como UPDATE, DELETE, INSERT. Dichas

operaciones se verán afectadas en su tiempo de ejecución, porque debe mantenerse una estructura de índices para la tabla en cada operación. Por lo tanto, se debe ser cauteloso en la implementación de índices para no afectar, en gran medida, el desempeño normal de la aplicación.

Por lo general, las columnas que forman los criterios de búsqueda en las cláusulas WHERE de las sentencias de consulta más críticas y utilizadas de la aplicación, son las mejores candidatas para ser indexadas. Sin embargo, otro factor importante por tomar en cuenta es la selectividad de dicha columna candidata por ser indexada. La selectividad es el porcentaje de registros concordantes dentro del total de registros de la tabla (Microsoft Corporation, 2009). Si el porcentaje es muy bajo, el índice será altamente selectivo, lo que conllevará a que éste elimine una gran cantidad de registros, y reducirá el tiempo necesario para retornar los registros resultantes. Pero si, por el contrario, el porcentaje de selectividad es muy alto, la implementación de un índice sobre dicha columna no resultará muy útil para mejorar el desempeño de la consulta SQL.

Para aquellos índices creados sobre múltiples columnas, debe tomarse en cuenta de que la columna con el mayor porcentaje de selectividad debe ser la columna que se encuentre del lado izquierdo (al inicio), del listado de columnas indexadas, ya que esto le permitirá al optimizador seleccionar el índice con mayor exactitud.

4.5 Restricciones de Integridad

Existen varias restricciones de integridad de datos que pueden implementarse en un DBMS. Entre ellas, están las restricciones declarativas NOT NULL, CHECK, FOREIGN KEY y PRIMARY KEY.

A continuación se dará énfasis a los efectos positivos de cada una de estas restricciones en el momento de diseñar una base de datos, ya que por razones específicas, algunos diseños no las implementan, reduciendo los beneficios que dichas restricciones aportan al DBMS.

4.5.1 NOT NULL

Las restricciones NOT NULL le permiten a un DBMS conocer, de antemano, que una columna no podrá almacenar valores NULL o NULOS. Si una columna almacena valores nulos, el espacio de almacenamiento para dicha columna será mayor, ya que la definición de un tipo de dato no incluye un valor reservado para la palabra clave NULL. Por lo tanto, el DBMS deberá reservar un espacio adicional, generalmente, al inicio del espacio de almacenamiento de la columna para indicar con una bandera si el valor es nulo o no (SQL Server utiliza un bit, Oracle un byte) (Gulutzan & Pelzer, 2002). Adicionalmente, el retornar una columna que almacene valores nulos, tomará más tiempo, ya que debe revisarse dicho indicador extra antes de poder procesar el valor que contiene la columna. Por lo tanto, para optimizar las sentencias SQL, es necesario definir como NO NULAS (NOT NULL) las columnas que así lo sean.

4.5.2 CHECK

Una forma muy eficaz y eficiente de validar los datos por almacenar en una columna, es el utilizar una restricción de integridad CHECK para asegurarse de que el dato por insertar obedece cierta regla impuesta. Por ejemplo, puede validarse fácilmente que la fecha de nacimiento de una persona esté dentro de un rango aceptable de años o inclusive evitar que se ingresen fechas de nacimiento a futuro. Dichas validaciones son procesadas por el DBMS de manera más eficiente y aseguran sobre todo la integridad de los datos almacenados en un lugar único para cualquier aplicación cliente.

4.5.3 FOREIGN KEY

Permite asegurar de que los datos referenciados a otras columnas que formen una llave primaria, existan de antemano. Lo anterior permite que se ingresen empleados a un departamento ya existente. Se debe ser muy cuidadoso a la hora de elegir si los cambios en las columnas padres deberán propagarse en cascada a las registros de las columnas hijas, ya que, eventualmente, la eliminación de un departamento podría hacer que un empleado sea eliminado también de la base de datos, lo que ciertamente causará serios problemas dentro de una organización.

4.5.4 PRIMARY KEY

Como primera regla acerca de una llave primaria o PRIMARY KEY: toda tabla debe tener una. Lo anterior es teoría básica relacional indiscutible. Aún siendo una tabla temporal, podrá crearse una llave primaria compuesta. Y, sin duda, el optimizador del DBMS hará uso de dicha información para hacer que las sentencias SQL se ejecuten de manera más rápida, sobre todo en situaciones en que deban extraerse datos de más de dos tablas.

4.6 Procedimientos Almacenados

Las ventajas de utilizar procedimientos almacenados, en orden de importancia son:

1. Los procedimientos almacenados residen en el servidor de la base de datos, así que los mensajes no necesitan ser enviados y recibidos hacia el cliente, durante el tiempo en que el procedimiento es ejecutado.

2. Los procedimientos almacenados son analizados sintácticamente (*parse*, en inglés) una única vez por el DBMS, y los resultados de dicho análisis son almacenados persistentemente, así que no existe necesidad de analizarlos nuevamente por cada ejecución.

3. Al acceso a la ejecución de los procedimientos almacenados, se protegen de la misma manera que cualquier otro objeto de la base de datos.

4. El código que representa las reglas del negocio residen en un solo lugar, así que la propagación de cambios en dichas reglas puede suceder de manera inmediata hacia los clientes que ejecuten los procedimientos almacenados, y sin necesidad de recompilar el código del cliente y actualizarlo (Gulutzan & Pelzer, 2002).

Con lo que el uso de procedimientos almacenados para aplicaciones comerciales, se asegura la consistencia y eficacia en la aplicación de lenguaje procedimental, así como se agrega un nivel extra de seguridad para acceder a su ejecución.

5 Conclusiones y Recomendaciones

En este artículo se recopila una pequeña muestra de las técnicas y recomendaciones que pueden aplicarse para mejorar el desempeño de la ejecución de sentencias SQL.

Siguiendo una serie de recomendaciones sencillas, se hizo evidente de que con algunos cambios menores en la manera de escribir una sentencia SQL, puede lograrse una mejoría en el desempeño de la ejecución de ésta. Además, se recalca la necesidad de entender el código escrito, antes de iniciar con el proceso de optimización, ya que un cambio evidente puede no ser el necesario en códigos SQL antiguos.

Para lograr una óptima ejecución de las sentencias SQL, es necesario complementar las técnicas y recomendaciones de sentencias SQL, con un adecuado y cuidadoso diseño de la base de datos, para lograr un mejor desempeño en la extracción de datos, como son la implementación de índices sobre columnas de búsqueda frecuentemente utilizadas.

Sin embargo, una de las mejores recomendaciones se enfoca, principalmente, a seguir el estándar ANSI-SQL para la definición de sentencias SQL, lo que, sin duda, aumenta la portabilidad del código escrito y mejora significativamente la claridad de las condiciones que conllevan su ejecución para lograr los resultados esperados, así como facilitan su fácil comprensión.

Además, se agregaron ciertas recomendaciones para asegurar la integridad de la información almacenada dentro de la base de datos, como son las restricciones de integridad que deben ser empleadas casi de manera obligatoria.

Otra recomendación importante radica en el uso de procedimientos almacenados para agilizar y unificar código procedimental que tenga estrecha relación con la manipulación de información en la base de

datos, con lo que, sin duda, se manejará de manera más eficiente operaciones sobre los datos, y de manera consistente sin afectar el código cliente de manera directa.

Sin duda, los mayores beneficiados serán los usuarios finales de las aplicaciones que se desarrollen, los cuales podrán experimentar un cierto grado de mejora en el uso de las mismas. Cualquier aplicación comercial, que busque aumentar el grado de optimización en el manejo y extracción de datos de un repositorio de bases de datos; sin duda, conllevará a una mayor aceptación dentro de un mercado cada vez más competitivo y selectivo.

6 Bibliografía

AskTom. (12 de agosto del 2001). *Difference between count(1) and count(*)*. Recuperado el 3 de marzo del 2009, de http://asktom.oracle.com/pls/asktom/f?p=100:11:3377074859490575::::P11_QUESTION_ID:1156159920245

Berlanga Llavori, R., Inesta Quereda, J. M., García Sevilla, P., Gracia Luengo, I., & Barber Miravalles, F. (2000). *Introducción a la programación con Pascal*. Castellón de la Plana, Valencia, España: Universitat Jaume I.

Bowman, J. S., Emerson, S. L., & Darnovsky, M. (2001). *The Practical SQL Handbook: Using SQL Variants.* Addison-Wesley.

Carter, B. (2004). *SQL Anywhere Studio 9.* Texas, Estados Unidos: Wordware Publishing, Inc.

Dam, S. (2004). *SQL Server Query Performance Tuning Distilled.* Berkeley, California, Estados Unidos: Apress.

Dixit, A. (1.° de mayo del 2008). *Tuning SQL Statements That Use the IN Operator*. Recuperado el 1.° de marzo de 2009, de Scribd: http://www.scribd.com/doc/2713891/Tuning-SQL-Statements-That-Use-the-IN-Operator

Gortari, E. (1988). *Diccionario de la lógica*. México: Plaza y Valdés.

Graham, C. (24 de julio del 2008). *Dataquest Insight: Relational Database Management System Software by Operating System Market Share Analysis, Worldwide, 2007*. Recuperado el 8 de febrero del 2009, de http://mediaproducts.gartner.com/reprints/ncr/article19/article19.html

Gulutzan, P., & Pelzer, T. (2002). *SQL Performance Tuning.* Boston, Massachusetts, Estados Unidos: Addison Wesley.

Kriegel, A., & Trukhnov, B. (2008). *SQL Bible* (Segunda ed.). Hoboken, New Jersey, Estados Unidos: John Wiley and Sons.

Microsoft Corporation. (2009). *Query Performance Tuning (SQL Server Compact)*. Recuperado el 15 de marzo del 2009, de http://msdn.microsoft.com/en-us/library/ms172984.aspx

Rob, P., Crockett, K., Crockett Rob, C., & Coronel, C. (2008). *Database Systems: Design, Implementation and Management.* Reino Unido: Cengage Learning EMEA.