VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

DESARROLLO Y APLICACIÓN DE TÉCNICAS NOVEDOSAS PARA EVALUAR EL DESEMPEÑO DE LA INSTRUMENTACIÓN DE FUNCIONES DE AGREGACIÓN EN SISTEMAS MANEJADORES DE BASES DE DATOS RELACIONALES

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA
(COMPUTACIÓN)

P R E S E N T A:

CHRISTIAN MENA RUVALCABA

DIRECTOR DE TESIS: M en C. JAVIER GARCÍA GARCÍA

MÉXICO, D.F. 2007.

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

A mi mama, por su apoyo y cariño incondicional y por estar siempre cuando más la he necesitado.

A mis hermanas, Miriam y Karla por su cariño y apoyo.

A mis profesores, por haber compartido un poco de sus conocimientos y paciencia.

A mi tutor Javier García, por compartir un poco de su conocimiento y ayudarme a ser un mejor profesionista.

A mis amigos, Rene Alejandro, Edgar Ricardo, Cesar Antonio, Israel, Eduardo, Gustavo y Maria del Angel por su apoyo y amistad.

Al laboratorio de bases de datos de la Facultad de Ciencias por su colaboración.

A CONACYT, por su apoyo brindado.

Al "Macroproyecto Tecnologías de la Información y la computación, para la Universidad", por su apoyo.

Por último a la UNAM, por brindarme los recursos necesarios para ser alguien en la vida.

Índice general

1. Introducción **7**

1.1. Contribución y relevancia 9

1.2. Objetivos 9

1.3. Trabajos relacionados 10

2. Tipos de almacenamiento en Sistemas Manejadores de Bases de Datos Relacionales **13**

2.1. Panorama general 13

2.2. Tipos de almacenamiento de datos 14

2.3. Tipos de organización de archivos 17

2.3.1. Archivos no ordenados 17

2.3.2. Archivos secuenciales u ordenados 18

2.4. Conceptos básicos sobre índices 20

2.5. Índices sobre archivos secuenciales 21

2.5.1. Índices multinivel 24

2.5.2. Índices primarios 25

2.5.3. Índices agrupados (Clustering indexes) 26

2.5.4. Índices secundarios 28

2.6. Índices en árboles-B^+ 29

3. Optimización de consultas y funciones de agregación 37

3.1. Algoritmos de ejecución de consultas 38

3.1.1. Algoritmos para la selección 39

3.1.2. Algoritmos para la reunión (join) 41

3.2. Implementación de las funciones de agregación 43

3.3. Procesamiento de las consultas 44

3.4. Heurísticas de optimización del álgebra relacional 47

3.5. Estimación de los tamaños de las operaciones 51

3.6. Ejecución de expresiones 55

3.7. Elección del plan de ejecución 56

4. Funciones Definidas por el Usuario (UDFs) 59

4.1. MySQL 60

4.1.1. UDFs escalares 60

4.1.2. UDFs agregadas 65

4.2. PostgreSQL 68

4.2.1. UDFs escalares 68

4.2.2. UDFs agregadas 75

4.3. SQL Server 2005 80

4.3.1. UDFs escalares 80

4.3.2. UDFs agregadas 88

4.4. Ventajas y desventajas de las UDFs 102

5. Experimentación 103

5.1. TPC-H 106

5.2. Funciones de agregación 107

5.3. Tipos de índices utilizados 109

5.4. Organización de los experimentos 109

5.5. Resultados 111

5.5.1. Funciones de agregación simples 111

5.5.2. Funciones de agregación complejas 128

6. Conclusiones 137

A. Consultas instrumentadas en la fase de experimentación 143

B. Ejemplos completos de UDFs agregadas 149

B.1. UDF para la correlación en MySQL 149

B.2. UDF para la suma en PostgreSQL 155

B.3. UDF para la correlación en PostgreSQL 156

B.4. UDF para la regresión lineal en SQL Server 2005 161

Bibliografía **162**

Capítulo 1

Introducción

Existen numerosos trabajos relacionados con la optimización de Sistemas Manejadores de Bases de Datos (SMBDs), desde el análisis de algoritmos para mejorar el desempeño de las consultas [13, 9, 33, 14], pasando por propuestas de estructuras para acelerar el acceso a los datos [23, 21, 12, 20] hasta propuestas de heurísticas para la obtención de planes de ejecución aceptables [5, 18, 29, 28], entre otras. Sin embargo, se ha estudiado poco la optimización de funciones de agregación. En el presente trabajo estudiaremos a profundidad el desempeño de funciones de agregación incluyendo el desempeño de funciones agregadas construidas mediante Funciones Definidas por el Usuario (UDFs por sus siglas en ingles), técnica que se pretende integrar como nueva dentro del paradigma de la teoría de la optimización.

Por otra parte, cuando se desea realizar experimentación sobre bases de datos relacionales se pueden seguir diversas técnicas para realizar lo anterior. En este trabajo se propone una alternativa novedosa para evaluar el desempeño de funciones de agregación en distintos SMBDs, esta alternativa de experimentación es útil cuando se quiere saber cual técnica para instrumentar funciones de agregación es la más eficiente, por ejemplo, en este trabajo deseamos investigar si las funciones de agregación programadas mediante UDFs pueden ser más eficientes que las construidas en SQL o las funciones intrínsecas en SQL (ejem. SUM()), las cuales son las dos técnicas tradicionales utilizadas al momento de implementar o construir una función de agregación.

Mostraremos que las UDFs pueden servir como una nueva técnica de optimización (para funciones de agregación) y en general veremos que se logra un mejor desempeño al que se obtiene en SQL. Aunado a esto, se utilizará la teoría de optimización de consultas para mostrar que la instrumentación de las UDFs no es independiente de las demás técnicas de optimización clásicas, sino que pueden implementarse en conjunto e incluso obtener mejores resultados.

Partimos con la teoría de optimización, la cual comprende: la teoría de índices, los algoritmos que realizan las operaciones de las consultas, las heurísticas del algebra relacional para transformar las expresiones de una consulta en otras más eficientes, entre otras. Todo esto a efecto de tener un tiempo de ejecución menor.

Posteriormente se expone uno de los puntos medulares de la presente investigación, el procedimiento para la construcción de funciones de agregación (y funciones escalares) a través de UDFs en tres manejadores de bases de datos muy populares: MySQL, PostgreSQL y SQL Server 2005.

En la etapa de la experimentación se explican los componentes utilizados como son: la base de datos sintética TPC-H [31] (esta herramienta nos proporciona una base de datos consistente y no sesgada sobre la cual podemos realizar pruebas), los sistemas operativos, las características del hardware utilizado, entre otras. Asimismo, se explica el conjunto de funciones de agregación utilizadas para mostrar la teoría antes mencionada.

Finalmente se instrumento todo lo anterior (teoría de la optimización y UDFs) en la base de datos sintética. Esto se realiza en los tres SMBDs previamente citados, con el fin de mostrar que las UDFs pueden ser consideradas como una nueva técnica de optimización, al mostrar un tiempo de ejecución generalmente menor que su contraparte en SQL para las funciones de agregación y utilizando las funciones de agregación intrínsecas de los SMBDs citados.

Asimismo, mostramos que la alternativa para la experimentación utilizada es adecuada para la realización de estudios experimentales del desempeño de funciones de agregación instrumentadas a través de diversas técnicas en distintos manejadores de bases de datos.

1.1. Contribución y relevancia

Pretendemos trabajar con aspectos novedosos como son las extensiones en los manejadores de bases de datos para programar UDFs agregadas, a efecto de mostrar que pueden integrarse como una nueva técnica dentro de la teoría de optimización en bases de datos relacionales.

Adicionalmente, pretendemos dar una alternativa novedosa para el estudio y experimentación sobre el desempeño de funciones de agregación que sean instrumentadas de diversas maneras en bases de datos relacionales a efecto de observar cual técnica resulta ser más eficiente, los resultados de esta investigación pueden ser utilizados en otras investigaciones que tengan que ver con sistemas manejadores de bases de datos y en las que se desee realizar evaluaciones experimentales del desempeño de las funciones de agregación. Esta alternativa para la experimentación se desarrolló durante la experiencia recabada en la elaboración de los trabajos [24, 26] y el trabajo [22], este último con la participación del autor de la presente tesis.

Todo lo anterior será implementado en tres manejadores de bases de datos ampliamente conocidos: MySQL, PostgreSQL y SQL Server2005 a efecto de mostrar los resultados experimentales.

1.2. Objetivos

Realizar investigación de vanguardia que permita incrementar, evaluar y extender el conocimiento actual sobre la instrumentación de las funciones de agregación construidas mediante UDFs, SQL y funciones intrínsecas en SQL.

Mostrar cual de estas técnicas es más eficiente al instrumentar funciones de agregación.

Mostrar que las UDFs pueden ser instrumentadas con las técnicas de optimización clásicas y mostrar los resultados de su desempeño.

Diseñar y aplicar una alternativa novedosa de experimentación para evaluar el desempeño de funciones de agregación instrumentadas mediante distintas

técnicas en SMBDs relacionales.

1.3. Trabajos relacionados

Hoy en día existen muchos trabajos referentes a bases de datos relacionales, asimismo, muchos de ellos atacan directamente la parte de optimización de consultas en bases de datos. Sin embargo, existen realmente muy pocos trabajos relacionados al desempeño de las funciones agregación, como veremos a continuación:

En [5] el autor nos da un panorama general de ciertas técnicas para optimizar consultas en un sistema de base de datos relacional, la evaluación de estas consultas las divide en dos componentes: el optimizador de consultas (encargado de representar la consulta SQL en una forma eficiente) y la forma de ejecución de la consulta (que algoritmo efectuará cada operación). Además, propone ciertos criterios para tener una buena optimización de consultas, asimismo, explica algunas técnicas de optimización utilizando reuniones (joins) como son: LEFT OUTER JOIN y RIGHT OUTER JOIN. Por último explica la utilización de vistas como técnica de optimización. En [23] los autores definen dos nuevas estructuras de índices para la evaluación eficiente de consultas estilo OLAP: los índices Bit-Sliced y los índices Projection, los cuales son una aproximación a los índices Bitmap. En [13] el autor examina como el operador JOIN puede ser eficientemente computado, enfatiza ciertas consideraciones que se tienen que tener al trabajar con este operador, además de hacer un estudio de ciertos algoritmos para el tratamiento de los JOINS (así como su costo medido en E/S-entradas/salidas a disco) cuando tenemos una llave primaria o ninguna. En [3] el autor examina las particularidades de los árboles de búsqueda binarios multidimensionales como una extensión de los árboles binarios de búsqueda tradicionales para el caso donde tenemos más de una llave de búsqueda. En [9] los autores exploran el desempeño de 4 algoritmos (Merge-sort, Hash simple, Hash-GRACE y Hash-Híbrido) utilizados para realizar operaciones en sistemas de bases de datos relacionales, aunado a esto, muestran las bondades de estos algoritmos al cumplirse ciertos criterios, asimismo, analizan sus respectivos costos de E/S a disco. En [27] los autores comparan experimentalmente las Funciones Definidas por el Usuario y SQL con respecto al desempeño (tiempo de ejecución), mostran-

do que las UDFs son más rápidas que las agregaciones estándar en SQL y tan rápidas como las expresiones aritméticas en SQL, esto lo realizaron en el SMBD Teradata. En [25] el autor hace un análisis profundo de las bondades que existen al utilizar las UDFs para la construcción de modelos estadísticos, al definir UDFs escalares y UDFs agregadas que permiten el manejo de dos matrices de estadísticas suficientes (sufficient statistical matrices), las cuales son fundamentales para ciertos modelos estadísticos, asimismo, explica como pueden ser integrados los modelos lineales multidimensionales dentro del SMBD Teradata con SQL y UDFs. En el presente trabajo no sólo se tratan las funciones de agregación estándar, sino además se proponen funciones de agregación complejas (estás funciones generalmente no están instrumentadas en los SMBDs tradicionales) implementándolas en tres SMBDs: MySQL, PostgreSQL y SQL Server 2005, asimismo, se instrumentan éstas en conjunto con otros mecanismos de optimización. En [7] la autora presenta un modelo matemático de las funciones de agregación, mostrando su correspondencia con las UDFs agregadas en SQL server 2005, Informix y Oracle. Asimismo, estudia estas funciones en vistas materializadas. En todos estos trabajos, no se estudia a detalle las funciones de agregación con la profundidad realizada en el presente estudio, tampoco atacan el problema de optimización a detalle como lo abordamos en este trabajo. Por último, en [22] los autores proponen la construcción e implementación de funciones de agregación complejas mediante UDFs en lenguaje C utilizando los SMBDs PostgreSQL y MySQL. Asimismo, analizan el desempeño de las UDFs en ambos SMBDs contra funciones de agregación construidas en SQL puro y funciones de agregación intrínsecas en SQL (ejem. SUM()). En el presente trabajo adicionalmente se utiliza el SMBD SQL Server 2005, y se instrumentan con diversas técnicas de optimización no estudiadas anteriormente.

En [4] los autores dan una metodología para poder medir el desempeño de distintos SMBDs relaciones a nivel multiusuario, utilizan varias consultas para este fin y aplican su metodología para medir el desempeño de INGRES y ORACLE, sin embargo, se enfocan en medir el desempeño de los SMBDs y no profundizan tanto en las funciones de agregación como en este trabajo, tampoco tratan diferentes arquitecturas de hardware y el número de técnicas de optimización utilizadas son menores. Cabe mencionar que actualmente existen varias metodológicas que prueban el desempeño de los SMBDs o de ciertos componentes , sin embargo, no existe una metodología de experimentación que permita evaluar el desempeño de instrumentar funciones

de agregación mediante distintas técnicas. Esto es lo que nuestra alternativa de experimentación trata de cubrir, misma que se construyó tratando de simular un ambiente común de trabajo al momento de utilizar funciones de agregación, también se integraron distintas distintas técnicas de optimización clásica para cubrir una mayor gama de casos.

Capítulo 2

Tipos de almacenamiento en Sistemas Manejadores de Bases de Datos Relacionales

2.1. Panorama general

La *teoría de la optimización* es el proceso de selección del plan de ejecución de consultas más eficiente de entre muchas estrategias generalmente disponibles para el procesamiento de una consulta dada, especialmente si la consulta es compleja. En general no esperamos que los usuarios escriban las consultas de modo que puedan procesarse de manera eficiente. Por el contrario, se espera que el sistema cree un plan de ejecución que minimice el costo de ejecución de las consultas. Bajo escenarios de esta naturaleza es donde entra la teoría de la optimización [1].

Un aspecto de la optimización de consultas tiene lugar al nivel del álgebra relacional, donde el sistema intenta hallar una expresión que sea equivalente a la expresión dada, pero con un tiempo de ejecución más eficiente. Otro aspecto es la elección de una estrategia detallada para el procesamiento de

la consulta, esto es, qué algoritmo se utilizará para ejecutar cada operación[1], la selección de los índices correctos que se van a emplear, etc. Asimismo, muchas de estas estrategias se utilizan cuando estudiamos el desempeño de funciones de agregación. Por esta razón, vale la pena estudiar a profundidad estas estrategias de optimización.

La diferencia en el costo (en términos de tiempo de ejecución) entre una estrategia buena y una mala suele ser sustancial, y puede resultar de varios órdenes de magnitud. Por lo tanto, vale la pena que el sistema invierta una cantidad importante de tiempo en la selección de una buena estrategia para el procesamiento de la consulta (o función de agregación).

Otra herramienta muy importante para la optimización de consultas son los índices, ya que proporcionan un camino a través del cual se pueden localizar y acceder a los datos de una manera más optima.

En este capítulo comenzamos con el análisis de los diferentes tipos de almacenamiento de datos, posteriormente estudiamos las diferentes organizaciones de archivos y finalmente estudiaremos a profundidad las estructuras de índices. En el Capítulo 3, nos enfocaremos al estudio de los planes de ejecución.

2.2. Tipos de almacenamiento de datos

Cualquier conjunto de datos debe ser almacenado físicamente para poder ser utilizado posteriormente cuando así se requiera, así los SMBDs pueden extraer, actualizar y procesar estos datos como se vayan necesitando. Los medios de almacenamiento se clasifican principalmente en dos tipos [11]:

- **Almacenamiento primario**. Esta categoría incluye los medios de almacenamiento que pueden ser operados directamente por la unidad central de procesamiento (CPU), tal como la memoria principal (RAM) y una más pequeña pero más rápida, la memoria cache. La memoria

[1] En este trabajo sólo analizaremos de manera general los distintos algoritmos que se pueden implementar, si se desea ver un estudio detallado véase [17].

principal ofrece un rápido acceso a los datos pero su capacidad de almacenamiento es limitado.

- **Almacenamiento secundario y terciario**. Esta categoría incluye a los discos magnéticos (discos duros), discos ópticos y cintas. Los discos magnéticos son considerados como almacenamiento secundario, mientras que los discos ópticos y las cintas como almacenamiento terciario. Todos estos mecanismos de almacenamiento tienen una mayor capacidad de almacenaje, un costo menor y proveen un acceso menor a los datos en comparación con los mecanismos de almacenamiento primario. Por otra parte, los datos en mecanismos de almacenamiento secundario y terciario no pueden ser procesados directamente por el CPU, primero se tienen que copiar dentro del almacenamiento primario. Siendo esta acción la más costosa como veremos posteriormente.

Los SMBDs generalmente almacenan grandes cantidades de datos que deben persistir durante grandes períodos de tiempo. Los datos son leídos y procesados repetidamente durante este período. Las bases de datos son guardadas permanentemente en discos magnéticos por las siguientes razones:

- Generalmente las bases de datos son muy grandes para caber completamente en memoria principal.
- Las circunstancias que causan una perdida permanente de datos es menos frecuente para los discos de almacenamiento secundario (almacenamiento no-volátil) que para el almacenamiento primario (almacenamiento volátil).
- El tiempo para acceder los datos desde una unidad de disco magnético es varios ordenes de magnitud mayor al tiempo de acceso desde memoria principal, sin embargo, el costo monetario relativo de almacenamiento es menor en disco que en memoria principal.

Ahora bien, lo siguiente a considerar es la forma en que los archivos de datos pueden ser organizados en el disco para un buen desempeño de las bases de datos. Las técnicas usadas para almacenar grandes cantidades de datos estructurados en disco son muy importantes para el diseñador de bases de

datos, el administrador de bases de datos (DBA) y usuarios de SMBDs. Los diseñadores y los DBAs deben conocer las ventajas y desventajas de cada una de las técnicas de almacenamiento al momento de diseñar, implementar y operar una base de datos sobre un SMBD especifico.

Generalmente las aplicaciones sobre bases de datos sólo requieren una pequeña porción de la misma al momento del procesamiento, siempre que una porción de los datos sea requerida éstos deberán ser localizados en el disco, copiados a memoria principal para su procesamiento respectivo y luego ser reescritos a disco si los datos han sido modificados. El almacenamiento de los datos en disco esta organizado como *archivos de registros*. Cada registro es una colección de valores de datos. Los registros deben ser guardados en disco de tal manera que sea posible localizarlos rápidamente cuando se les necesite.

Clasificaremos las organizaciones de archivos en dos tipos[2]: Las **organizaciones de archivos primarias y las secundarias**. Las primeras determinan cómo el registro de archivos será colocado físicamente en disco, asimismo, determinan cómo los registros pueden ser accedidos. Un *archivo no ordenado* (heap file) coloca los registros en disco sin un orden en particular colocando un nuevo registro al final del archivo. En cambio un *archivo ordenado* (sorted file) o archivo secuencial, mantiene los registros ordenados por el valor de un cierto atributo o campo llamado *llave ordenada* (sort key). Existen otras estructuras de archivos tales como los "*árboles-B*" las cuales usan estructuras de árbol. Por otro lado, una organización secundaria permite el acceso eficiente al registro de archivos basado en campos alternativos a los utilizados por la organización de archivos primarios, este último tipo de organización no será tratada en el presente trabajo.

Los datos suelen ser almacenados en forma de registros, cada registro consiste en una colección de valores, donde cada valor esta formado por uno o más bytes. Los registros de un archivo deben ser colocados en *bloques de disco* debido a que un bloque es la unidad de transferencia de datos entre el disco y la memoria principal. Cuando el tamaño del bloque es mayor al tamaño del registro, cada bloque contendrá numerosos registros, para el caso contrario, se puede guardar parte del registro en un bloque y el resto en otro (o en varios). Un indicador (puntero) al final del primer bloque señala al bloque

[2]En el presente trabajo se tomó el sistema de archivos por defecto por cada SMBD, debido a que suponemos que la base de datos ya esta creada.

que contiene el resto del registro en caso de no ser el bloque consecutivo en el disco y así sucesivamente.

Hay algunas técnicas estándar para asignar los bloques de un archivo a disco. En la *asignación contigua*, los bloques de archivo son asignados a bloques de disco consecutivos, esto hace la lectura de todo el archivo muy eficiente, pero hace difícil la expansión del archivo. En la *asignación enlazada*, cada bloque de archivo contiene un puntero al siguiente bloque de archivo. Esto permite expandir el archivo fácilmente, pero provoca que la lectura completa del archivo sea lenta. Otra posibilidad es la *asignación indexada*, donde uno o más bloques de índices contienen un apuntador al bloque de archivo actual. También es común el utilizar una combinación de estas técnicas [11].

2.3. Tipos de organización de archivos

En esta sección discutiremos algunos métodos para organizar los registros de archivo en disco. Veremos algunas ventajas y desventajas de utilizar cada una de estas alternativas.

2.3.1. Archivos no ordenados

Este es el tipo más simple de organización de archivos, los registros son colocados en el archivo de acuerdo al orden en el cual fueron ingresados, entonces un nuevo registro es insertado al final del archivo. Esta organización es a veces utilizada en compañía de mecanismos de acceso adicionales, tales como los índices secundarios.

La **inserción** de un nuevo registro es muy eficiente, el último bloque de disco del archivo es copiado a un buffer, el nuevo registro es añadido y luego el bloque es nuevamente escrito a disco. Sin embargo, la **búsqueda** de un registro utilizando cualquier condición de búsqueda involucrará una búsqueda lineal en todos los bloques de archivo, un procedimiento costoso cuando nuestro archivo es considerablemente grande. Si sólo un registro satisface la condición de búsqueda, entonces en promedio, un programa accederá a

memoria y buscará en la mitad de los bloques de archivo antes de encontrar el registro deseado. Para un archivo de B bloques, se requerirá de buscar en $(B/2)$ bloques en promedio. Si el registro no se encuentra dentro del archivo, entonces el programa leerá y buscará en todos los B bloques de archivo.

Para **borrar** un registro, un programa debe primero encontrar el bloque donde esta contenido el registro, luego copia el bloque dentro de un buffer, se borra el registro del buffer y finalmente se reescribe el bloque a disco. Esto deja espacio no utilizado en el bloque del disco, entonces si borramos una gran cantidad de registros de esta forma, estaríamos desperdiciando espacio en disco. Otra técnica utilizada para borrar registros es tener un bit o byte extra, llamado *marcador de borrado*, el cual es guardado con cada registro. Un registro es borrado al poner el marcador de borrado en un cierto valor (ejem. 0 ó 1), un valor del marcador de borrado distinto al definido indica un registro valido (no borrado). Ambas técnicas de borrado requieren de una reorganización periódica del archivo para recuperar el espacio no utilizado provocado por los registros borrados. Durante la reorganización, los bloques de archivo son accedidos frecuentemente y re-empacados en un bloque quitando los registros borrados. Otra posibilidad es usar el espacio del registro borrado (marcado como borrado) cuando insertamos un nuevo registro, aunque esto requerirá de tener una mayor contabilidad de las pistas vacías así como de su localización.

2.3.2. Archivos secuenciales u ordenados

Un archivo secuencial como su nombre lo indica, es un archivo ordenado físicamente en disco, donde este ordenamiento esta basado en los valores de uno de los campos del archivo, llamado *campo de ordenamiento*. Esto conduce a un archivo secuencial. Si el campo de ordenamiento es también un campo llave del archivo, entonces el campo garantiza tener un único valor en cada registro, este campo es llamado *llave de ordenamiento* del archivo. La Figura 2.1 muestra un archivo secuencial con la CURP como su llave de ordenamiento (recordemos que la CURP es única para cada persona).

Tener los registros ordenados tiene ciertas ventajas sobre los archivos no ordenados. Primero, leer los registros de acuerdo a la llave de ordenamiento

	CURP	Nombre	Fecha_Nacimiento	Trabajo	Salario	Sexo
Bloque 1	ABCD121618xxx					
	BCDE111283xxx					
	CDEF100378xxx					

	CURP	Nombre	Fecha_Nacimiento	Trabajo	Salario	Sexo
Bloque 2	DEFG091179xxx					
	EFGH010779xxx					
	FGHI231182xxx					

	CURP	Nombre	Fecha_Nacimiento	Trabajo	Salario	Sexo
Bloque 3	GHIJ171128xxx					
	HIJK02083xxx					
	IJKL090876xxx					

. . .

	CURP	Nombre	Fecha_Nacimiento	Trabajo	Salario	Sexo
Bloque n-1	JKLM030373xxx					
	KLMN04064xxx					
	LNMO030507xxx					

	CURP	Nombre	Fecha_Nacimiento	Trabajo	Salario	Sexo
Bloque n	NMOP091180xxx					
	MERC121681xxx					
	ZORD171279xxx					

Figura 2.1: Ejemplo de un archivo secuencial con llave de ordenamiento (el campo CURP).

resulta ser eficiente, esto porque no se requiere de una fase de ordenamiento. Segundo, encontrar el siguiente registro a partir del registro actual de acuerdo a la llave de ordenamiento usualmente no requiere acceder nuevamente a un bloque adicional, esto porque el siguiente registro estará en el bloque actual (a menos que el registro actual sea el último del bloque). Tercero, una búsqueda basada en el valor de una llave de ordenamiento resulta en un acceso más eficiente cuando utilizamos una técnica de búsqueda binaria, la cual es más eficiente que una búsqueda lineal[3].

La **inserción** y el **borrado** de registros son operaciones costosas para un archivo secuencial porque los registros deben permanecer ordenados físicamente. Para insertar un registro, debemos encontrar su posición correcta en el archivo, basado en su campo de ordenamiento y luego hacer espacio en el archivo para poder insertar el registro en esa posición. Para un archivo grande esta tarea puede tomar mucho tiempo, en promedio, la mitad de los registros del archivo deberán ser movidos para hacer espacio suficiente para el nuevo registro. Esto significa que la mitad de los bloques de archivo deben

[3]Recordemos que la complejidad para una búsqueda binaria es del orden de $O(log(n))$

ser leídos y reescritos después de que los registros han sido movidos entre ellos. Para el borrado de registros, el problema es menos severo si utilizamos los marcadores de borrado y una reorganización periódica del archivo. Una opción para hacer más eficiente la inserción es mantener una pequeña cantidad de espacio en cada bloque para nuevos registros, sin embargo, una vez que éste se haya utilizado por completo, volveríamos al problema original. Otro método frecuentemente utilizado es crear un archivo temporal no ordenado, denominado *archivo de desbordamiento* (overflow). En esta técnica, el archivo secuencial actual es llamado archivo principal, entonces los nuevos registros son insertados al final del archivo de desbordamiento en lugar de su posición correcta en el archivo principal. Periódicamente, el archivo de desbordamiento es ordenado y fusionado con el archivo principal durante la reorganización. Con esta técnica la inserción llega a ser muy eficiente, pero el costo de la complejidad para los algoritmos de búsqueda se ve incrementado. Una búsqueda lineal debe ser realizada en el archivo de desbordamiento si después de una búsqueda binaria el registro no fue encontrado en el archivo principal.

Los archivos secuenciales son raramente utilizados en aplicaciones de bases de datos a menos que se utilicen en conjunción con un mecanismo de acceso adicional, llamados *índices primarios*, esto resulta en un archivo secuencial indexado.

Otras estructuras de datos pueden ser utilizadas como organización de archivos, por ejemplo tenemos a los *árboles-B*, los cuales discutiremos cuando estudiemos las estructuras de índices.

2.4. Conceptos básicos sobre índices

Definición 2.1. Un *índice* de una base de datos es una colección de elementos que permiten un rápido acceso a los registros que conforman dicha base de datos. Al aumentar drásticamente la velocidad de acceso a los elementos, se suelen usar sobre aquellos campos sobre los cuales se hagan frecuentes búsquedas. Dicho índice tiene un funcionamiento similar al índice de un libro, guardando parejas de elementos: el elemento que se desea indexar y su posición en la base de datos. Para buscar un elemento que este indexado,

sólo hay que buscar en el índice dicho elemento para, una vez encontrado, devolver el registro que se encuentre en la posición marcada por el índice [34].

En general hay dos tipos básicos de índices:

- **Índices secuenciales**. Estos índices están basados en una disposición ordenada de los valores.
- **Índices asociativos (hash)**. Estos índices están basados en una distribución uniforme de los valores a través de una serie de cubetas (buckets). El valor asignado a cada cubeta esta determinado por una función, llamada *función hash*.

Se considerarán varias técnicas de indexación. Ninguna de ellas es la mejor, sin embargo, para cada aplicación especifica de bases de datos existe una técnica más apropiada.

2.5. Índices sobre archivos secuenciales

Empezamos nuestro estudio de las estructuras de índices con la que es probablemente la más simple: De un *archivo secuencial*, llamado el archivo de datos, se extrae otro archivo, llamado archivo indexado. Ambos archivos están enlazados por pares de llaves-punteros. Una *llave de búsqueda* K en el archivo indexado esta asociada con un puntero de un registro del archivo de datos, el cual tiene la llave de búsqueda K. Esta organización es especialmente útil cuando la llave de búsqueda es la *llave primaria* de la relación, aunque no es así necesariamente. Cuando el índice corresponde a la llave primaria se dice que es un *índice primario*, los índices cuyas llaves de búsqueda especifican un orden diferente al orden secuencial del archivo se llaman *índices secundarios*.

A los archivos con un índice primario según la llave de búsqueda se llaman *archivos secuenciales indexados*. Representan uno de los esquemas de índices

más utilizados en los SMBDs. Se utilizan en aquellas aplicaciones que demandan un procesamiento secuencial del archivo completo, así como un acceso directo a sus registros.

Índices densos

Ahora que tenemos nuestros registros ordenados, podemos construir sobre ellos un *índice denso*, el cual es una secuencia de bloques que mantienen sólo las llaves de los registros y apuntadores que señalan a los registros completos. El índice es llamado denso porque cada llave del archivo de datos esta representada en el índice. Los bloques indexados del índice denso mantienen las llaves en el mismo orden que el archivo de datos. De aquí que las llaves y los punteros ocupan mucho menos espacio que los registros completos (véase la Figura 2.2 para aclarar lo anterior). Los índices son especialmente eficientes bajo aplicaciones de esta naturaleza, dado que quizá el archivo de datos no quepa completamente en memoria principal, pero muy probablemente si el archivo indexado.

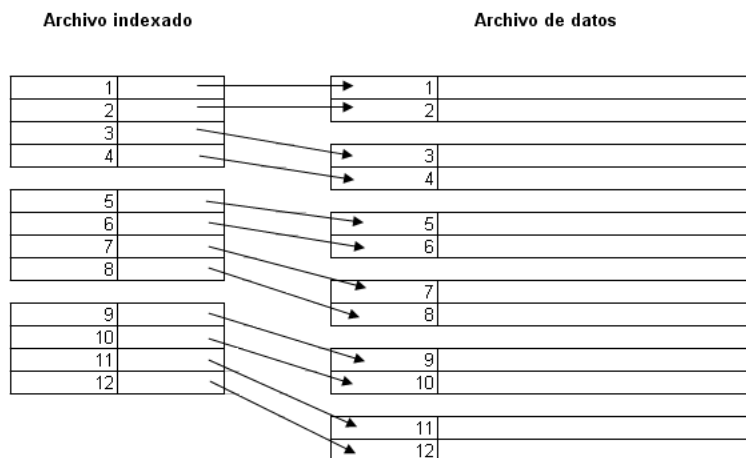

Figura 2.2: Ejemplo visual de un índice denso.

Como podemos observar en la Figura 2.2 nuestros bloques indexados sólo pueden mantener 4 pares de llaves-punteros. Vemos que el primer registro del archivo indexado apunta al primer registro del archivo de datos, el segundo al siguiente y así sucesivamente. Sin embargo, hay ocasiones en las que el

archivo indexado puede resultar ser muy grande debido a que mantiene una llave por cada registro del archivo de datos, para solventar este problema se suelen utilizar los *índices dispersos* que veremos más adelante.

Los índices densos soportan consultas que solicitan registros con un cierto valor de la llave de búsqueda. Dado un valor K de la llave de búsqueda, se busca el bloque indexado para K, y cuando se encuentra, se sigue el puntero asociado al registro completo con la llave K. Esto pudiese sugerir que se requiere examinar cada bloque del índice, o la mitad de los bloques del índice, en promedio, antes de encontrar K. Sin embargo, hay algunos factores que hacen la búsqueda basada en índices más eficiente de lo que parece.

- El número de bloques de índices es frecuentemente más pequeño comparado con el número de bloques de datos.
- Las llaves están ordenadas, podemos utilizar una búsqueda binaria para encontrar K. Si hay n bloques de índices, sólo tendremos que examinar $log_2 n$ bloques.
- El índice puede ser lo suficientemente pequeño para caber completamente en memoria principal. Si estamos bajo este escenario la búsqueda para la llave K involucraría sólo acceder a memoria principal, y no hay que hacer ninguna E/S a disco extra.

Índices dispersos

Cuando tenemos un índice muy grande, podemos utilizar una estructura similar al índice denso con una pequeña variante, llamado *índice disperso*, el cual utiliza menos espacio que el índice denso pero tarda un poco más en la búsqueda de un registro dada su llave. Un índice disperso, como vemos en la Figura 2.3, mantiene sólo un puntero llave por cada bloque de datos.

Para encontrar el registro con la llave K, dado un índice disperso, se busca el elemento del índice correspondiente a la llave máxima menor o igual que K. El archivo indexado esta ordenado por la llave, una búsqueda binaria localizará el valor K dentro del archivo indexado. Luego se sigue el puntero

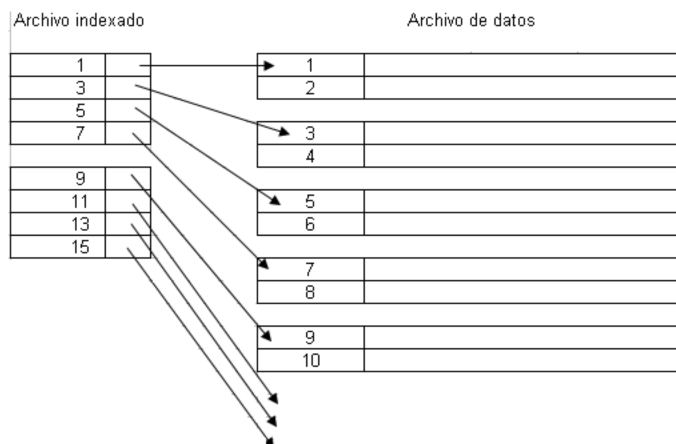

Figura 2.3: Ejemplo visual de un índice disperso.

asociado al bloque de datos, una vez en esta posición, se busca el registro con la llave K (usualmente con una búsqueda lineal).

2.5.1. Índices multinivel

Un índice puede referirse a muchos bloques, aún cuando se utilice una búsqueda binaria para obtener la llave K deseada, implicando tener que realizar muchas E/S a disco para obtener el registro deseado. Con el objeto de realizar de una forma más eficiente la búsqueda se puede crear un índice sobre otro índice, se puede agregar no sólo otro nivel, sino un tercer nivel adicional, sin embargo, es preferible utilizar una estructura en *árbol-B*, en lugar de sólo agregar niveles de índices.

En los índices multinivel, el primer nivel puede ser disperso o denso, sin embargo, a partir del segundo nivel, los demás deben ser dispersos. La razón es que un índice denso sobre un índice (cualquiera) tendría exactamente la misma cantidad de pares de llaves-punteros como el primer nivel, y por tanto, tomaría la misma cantidad de espacio como el tomado por el primer nivel [17]. Entonces un índice denso como segundo nivel no introduce ninguna ventaja, al contrario, sólo desperdiciaría espacio en disco. Para aclarar lo anterior veamos la Figura 2.4.

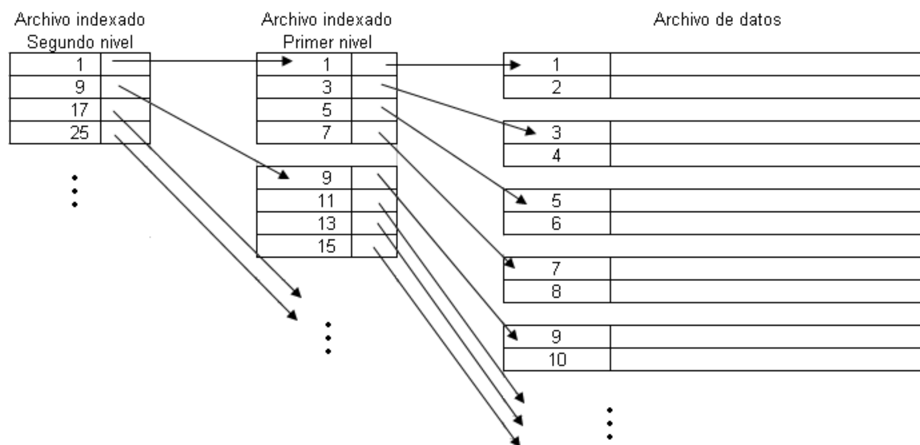

Figura 2.4: Ejemplo visual de un índice multinivel.

2.5.2. Índices primarios

Un *índice primario* es un archivo secuencial cuyos registros tienen dos campos, el primer campo es para el valor de la llave de ordenamiento, llamada *llave primaria* del archivo de datos, y el segundo campo es un apuntador a un bloque de direcciones. Hay un registro indexado en el archivo indexado por cada bloque en el archivo de datos. Cada registro del índice tiene el primer valor de la llave primaria de cada bloque y un puntero a este bloque como su segundo campo. El número total de entradas en el índice coincide con el número de bloques del archivo de datos. El primer registro en cada bloque del archivo de datos es llamado *registro ancla* del bloque. Un índice primario es un índice disperso, debido a que incluye un registro ancla por cada bloque del archivo de datos y no por cada valor del mismo. El archivo indexado para un índice primario necesita sustancialmente menos bloques que el archivo de datos. Por lo tanto, una búsqueda binaria sobre el archivo indexado requiere una menor cantidad de accesos a bloques. La Figura 2.5 muestra un archivo indexado que apunta al primer registro de datos de cada bloque, los cuales están ordenados y en donde el campo CURP es la llave primaria del archivo de datos.

Un gran problema con un índice primario (como cualquier archivo secuencial) es la inserción y el borrado de registros. Con un índice primario, el problema

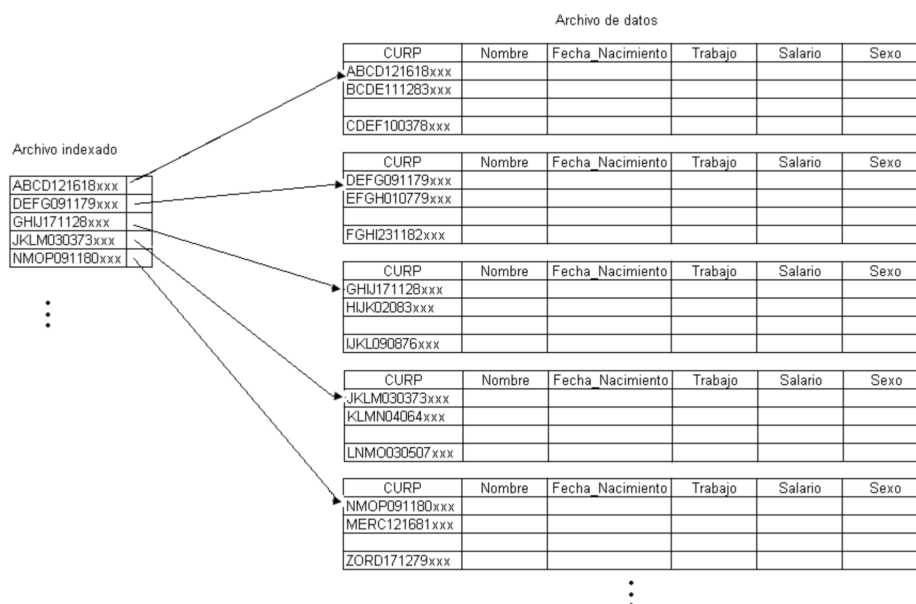

Figura 2.5: Ejemplo visual de un índice primario.

es complejo debido a que si intentamos insertar un registro en su posición correcta en el archivo de datos, debemos no sólo desplazar los registros para hacer espacio para el nuevo registro, sino además debemos cambiar los registros indexados, esto por los desplazamientos hechos en el archivo de datos los cuales cambiaran los registros ancla de varios bloques. Usando un archivo de desbordamiento (vistos en la Sección 2.3.2) podemos reducir este problema. El borrado de registros es tratado utilizando marcadores de borrado.

2.5.3. Índices agrupados (Clustering indexes)

Si los registros del archivo están físicamente ordenados de acuerdo a un campo no llave el cual no tiene un valor distinto para cada registro, tenemos entonces un *campo agrupado*. Podemos crear un tipo diferente de índice, llamado *índice agrupado*, el cual es útil para agilizar la extracción de registros que tienen el mismo valor en el campo agrupado. Esto difiere de un índice primario, el cual exige que el campo de ordenamiento del archivo de datos tenga un valor distinto para cada registro.

Un índice agrupado es también un archivo secuencial con dos campos; el primer campo es del mismo tipo que el campo agrupado del archivo de datos, y el segundo campo es un puntero. Hay una entrada en el índice agrupado por cada valor distinto del campo agrupado, la cual contiene el valor y un apuntador al primer bloque en el archivo de datos que tiene un registro con ese valor para su campo agrupado. La Figura 2.6 muestra un ejemplo de un índice agrupado.

Hay que notar que la inserción y el borrado todavía causan problemas debido a que los registros de datos están físicamente ordenados. Para solucionar el problema de la inserción, es común reservar un bloque completo (o un grupo de bloques contiguos) por cada valor del campo agrupado; todos los registros con ese valor serán colocados en ese bloque (o bloque agrupado). Esto hace la inserción y el borrado más eficientes.

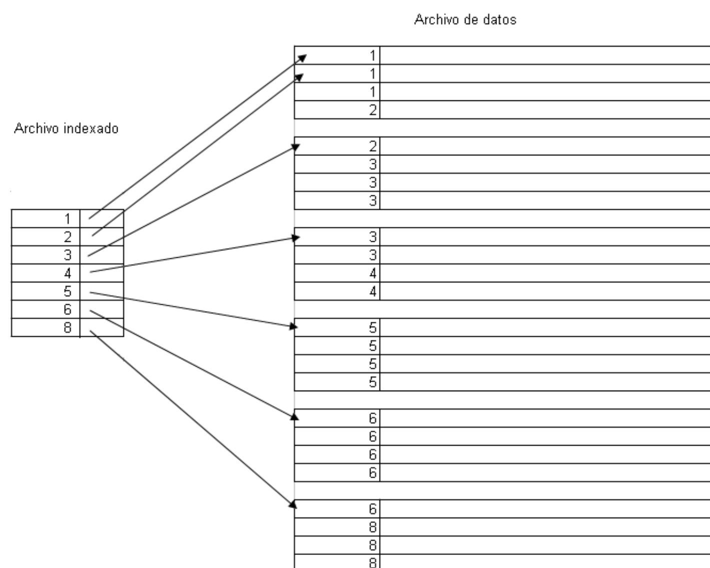

Figura 2.6: Ejemplo visual de un índice agrupado.

Un índice agrupado es otro ejemplo de un índice disperso, porque tiene una entrada por cada valor distinto del campo indexado, el cual es no llave por definición y por esto puede tener duplicados en lugar de un único valor por cada registro en el archivo.

2.5.4. Índices secundarios

Un *índice secundario* es un índice denso, usualmente con duplicados. Este índice consiste de pares de llaves-punteros; la llave es una llave de búsqueda y no necesariamente única. Los pares en el archivo indexado son ordenados por el valor de la llave, esto para ayudar a encontrar la entrada para una llave dada. Una característica importante a diferencia del índice primario es que el campo llave del archivo de datos no esta ordenado e inclusive puede tener valores duplicados.

Consideraremos un índice secundario sobre un campo llave que puede tener un valor distinto por cada registro. Tal campo a veces es llamado *llave secundaria*, en este caso hay un registro indexado por cada registro en el archivo de datos. Esto debido a que los registros del archivo de datos no están ordenados físicamente por los valores de la llave secundaria, entonces no podemos utilizar los registros ancla, como en el caso del índice primario. Para ver un ejemplo de esto véase la Figura 2.7.

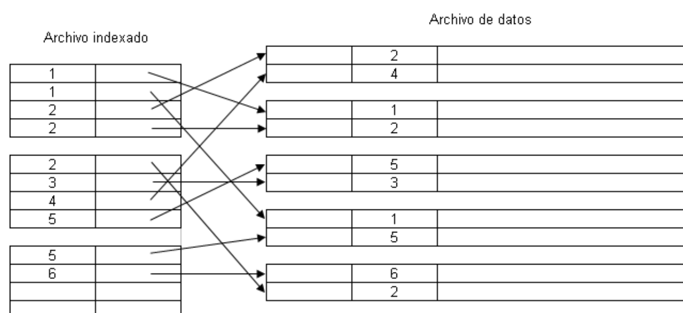

Figura 2.7: Ejemplo visual de un índice secundario.

Un índice secundario frecuentemente necesita más espacio de almacenamiento y más tiempo de búsqueda que un índice primario, esto debido a su mayor número de registros indexados. Para reducir un poco el tiempo de búsqueda cuando la llave de búsqueda aparece n veces en el archivo de datos, se suele utilizar un *nivel de indirección*, para únicamente escribir una sola vez cada valor distinto del archivo de datos y así evitar tener que escribir tantas veces en el archivo indexado como veces aparezca la llave de búsqueda (ejem. la Figura 2.7). Como mencionamos, una forma conveniente de evitar valores repetidos es utilizar un nivel de indirección (dividido en *cubetas*) entre el

archivo indexado secundario y el archivo de datos. Como se muestra en la Figura 2.8, los registros indexados apuntan a los registros con valores distintos de las cubetas en el nivel de indirección, las cuales a su vez tienen los valores ordenados, asimismo, estos registros (de las cubetas) apuntan a los registros del archivo de datos.

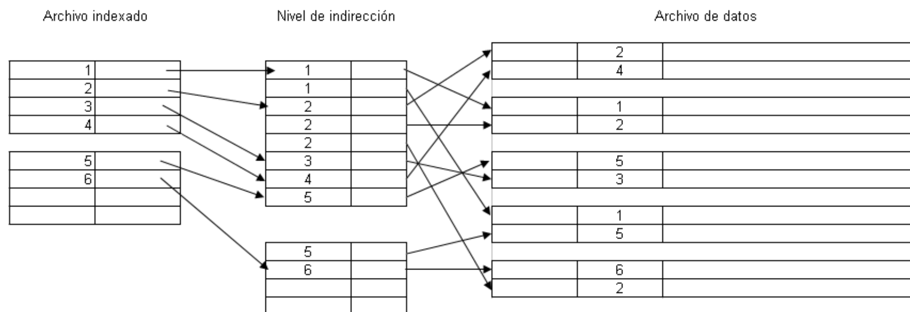

Figura 2.8: Ejemplo visual de un índice secundario con nivel de indirección.

Con esto concluimos nuestro estudio sobre las estructuras más simples de índices, a continuación presentamos algunas estructuras más sofisticadas y generalmente mucho más eficientes a las presentadas hasta ahora.

2.6. Índices en árboles-B$^+$

Mientras que uno o dos niveles son frecuentemente útiles al mejorar el desempeño de las consultas, hay una estructura más general la cual es la mayormente instrumentada en los SMBDs actuales. Esta estructura es llamada *árboles-B* (B-trees) [2] y una variante de ésta, llamada *árboles-B$^+$* (B$^+$-trees). A continuación estudiaremos los árboles-B$^+$ por ser los más ampliamente utilizados.

En general los árboles-B$^+$:

- Mantienen automáticamente tantos niveles de índices como sean apropiados de acuerdo al tamaño del archivo a indexar.

- Administran el espacio sobre los bloques de tal forma que cada bloque este a lo sumo medio lleno, por consiguiente un archivo de desbordamiento no será necesario.

Estas estructuras (árboles-B^+) implican una degradación del rendimiento al insertar y al borrar, además de un espacio extra. Este tiempo adicional es aceptable incluso en archivos con altas frecuencias de modificación, ya que se evita el costo de reorganizar el archivo. Asimismo, puesto que los nodos podrían estar a lo sumo medio llenos se desperdicia un mínimo de espacio, mismo que es aceptable dados los beneficios en el rendimiento aportados por estas estructuras [1].

Estructuras de los árboles-B^+

Brevemente estudiamos las estructuras de datos en árbol para introducir la terminología que utilizaremos posteriormente.

Un *árbol* esta formado por *nodos*, cada nodo en el árbol, excepto para un nodo especial, llamado *raíz*, tienen un nodo padre y algunos (cero o muchos) nodos *hijos*. El nodo raíz no tiene padre. Un nodo el cual no tiene ningún nodo hijo es llamado nodo *hoja*; un nodo "no hoja"es llamado un nodo *interno*. El *nivel* de un nodo es siempre uno más que el nivel de su padre, siendo el nivel del nodo raíz cero. Un *subárbol* de un nodo consiste en ese nodo y todos sus nodos descendientes (sus nodos hijos). Veamos la Figura 2.9 para aclarar todo lo anterior.

Usualmente se muestra un árbol con el nodo raíz en la cima, como se ve en la Figura 2.9. Una forma de representar un árbol es tener tantos punteros en cada nodo como hijos hayan en ese nodo.

Una vez que hemos analizado de manera general las estructuras en árbol, pasamos a estudiar las estructuras de los árboles-B^+. Los árboles-B^+ organizan sus bloques dentro de un árbol. El árbol es *balanceado*, esto significa que todos los caminos desde la raíz a una hoja tiene la misma longitud. Frecuentemente, hay tres capas o niveles en un árbol-B^+: la raíz, una capa intermedia, y las hojas, aunque cualquier número de niveles es posible.

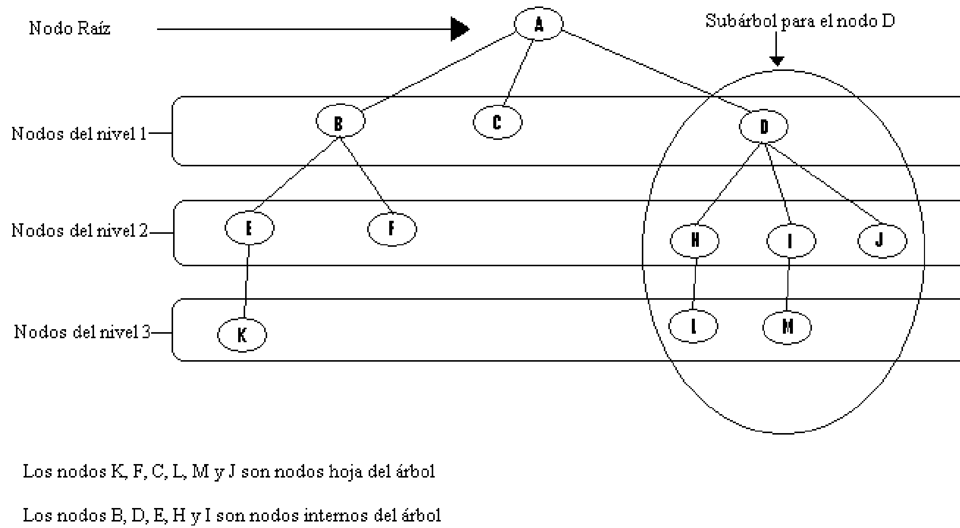

Figura 2.9: Ejemplo visual de una estructura en árbol.

Hay un parámetro n asociado con cada índice en árbol-B$^+$, y este parámetro determina la capacidad de almacenamiento para todos los bloques del árbol-B$^+$. Cada bloque tendrá el espacio para n valores de la llave de búsqueda y $n+1$ punteros. Un bloque en un árbol-B$^+$ es similar a los bloques indexados introducidos en la Sección 2.3, excepto que los bloques en un árbol-B$^+$ tienen un puntero extra, junto con los n pares de llaves-punteros. n será tan grande como llaves-punteros quepan en un bloque.

Hay algunas reglas importantes que se aplican a los bloques de un árbol-B$^+$:

- Las llaves en los nodos hojas son copias de las llaves del archivo de datos. Estas llaves están distribuidas a lo largo de las hojas de una manera ordenada, de izquierda a derecha.

- En el nodo raíz, hay al menos dos punteros utilizados[4]. Todos los punteros apuntan a los bloques del árbol-B$^+$ de un nivel inferior.

[4]Ignorando el caso cuando el nodo raíz es el único nodo en el árbol-B$^+$.

- En un nodo interior, todos los $n+1$ punteros pueden ser utilizados para apuntar a los bloques del árbol-B$^+$ de un nivel inferior. Al menos $\lceil \frac{n+1}{2} \rceil$ de estos punteros serán utilizados. Supongamos que j punteros son utilizados, entonces habría $j-1$ llaves, digamos $K_1, K_2, ..., K_{j-1}$. El primer puntero apunta a alguna parte del árbol-B$^+$ donde están los registros con las llaves menores a K_1. El segundo puntero va a la parte del árbol donde todos los registros cuyas llaves son al menos K_1, pero menores que K_2 y así sucesivamente. Finalmente el j-esimo puntero nos dirige a la parte del árbol-B$^+$ donde se encuentran los registros cuyas llaves son mayores o iguales a K_{j-1}.

- En los nodos hoja, el último puntero apunta al siguiente bloque hoja a la derecha, esto es, al bloque con los valores llave más grandes. Entre los otros n punteros en un bloque hoja, al menos $\lfloor \frac{n+1}{2} \rfloor$ de esos punteros serán utilizados para apuntar a los registros de datos, por lo tanto también al menos $\lfloor \frac{n+1}{2} \rfloor$ llaves serán necesarias en cada hoja. Los punteros que no son utilizados pueden ser tratados como nulos y no apuntaran a ningún lugar. El i-esimo puntero (si éste es utilizado) apunta a un registro con la llave i-esima.

Para aclarar lo anterior veamos la Figura 2.10. En esta Figura, $n = 3$, esto es, los bloques tienen capacidad para tres llaves y cuatro punteros. Vemos que el nodo raíz divide al árbol-B$^+$ en dos subárboles (los nodos que son menores a la raíz y los que son mayores o iguales a ella). En los nodos interiores (explicaremos el nodo interno del subárbol derecho) vemos que el primer puntero apunta a la parte del árbol-B$^+$ desde el cual se puede llegar a los registros que son menores que 23. El segundo puntero apunta a aquella parte del árbol-B$^+$ donde todos los registros cuyas llaves están entre las llaves del primer y el segundo bloque del árbol-B$^+$ ($23 \leq K < 31$). El tercer puntero es para todos los registros cuyas llaves estén entre el segundo y tercer bloque ($31 \leq K < 43$). Por último, el cuarto puntero nos permite alcanzar a los registros con llaves mayores o iguales a la llave del tercer bloque ($K \geq 43$). Ahora bien, en la misma Figura 2.10 podemos observar que en los nodos hoja, los primeros tres punteros van a los registros con esas llaves. El último puntero, como es siempre el caso para las hojas, apuntan a la siguiente hoja a la derecha, esto sería nulo si esta hoja fuera la última en la secuencia [17].

Como vemos el árbol-B$^+$ es una herramienta muy poderosa para construir

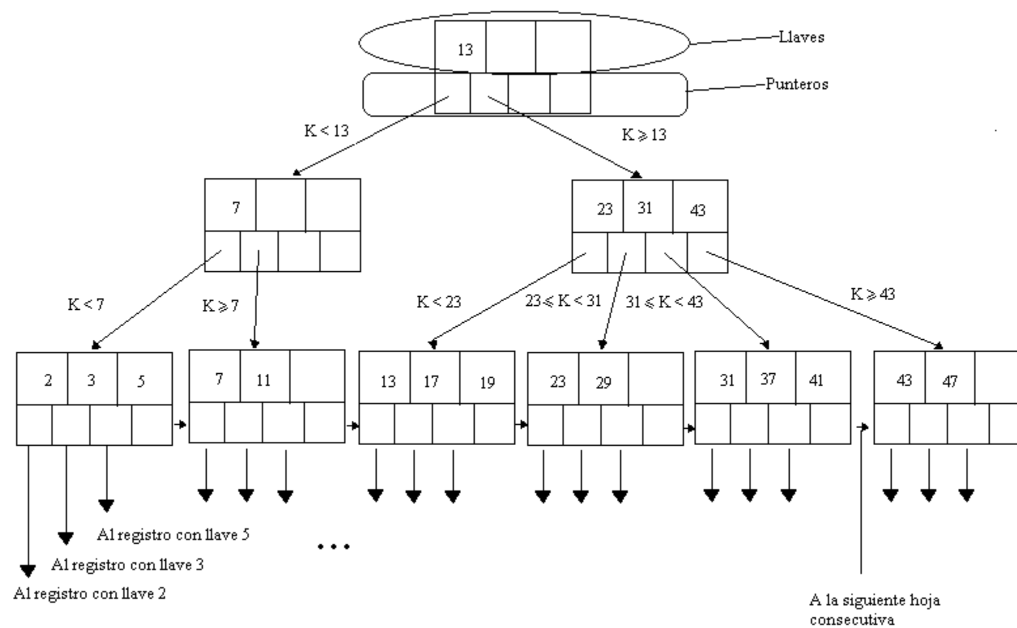

Figura 2.10: Ejemplo visual de un árbol-B$^+$.

índices. La secuencia de punteros de las hojas a los registros puede jugar el rol de varios tipos de índices, los cuales estudiamos en la Sección 2.5. Aquí mostramos algunos ejemplos:

- La llave de búsqueda es la llave primaria del archivo de datos, y el índice es denso. Esto es, hay un par de llaves-punteros en un nodo hoja para cada registro del archivo de datos. El archivo puede o no estar ordenado por la llave primaria.
- El archivo de datos esta ordenado por sus llaves primarias, y el árbol-B$^+$ es un índice disperso con un par de llaves-punteros en un nodo hoja por cada bloque del archivo de datos.
- El archivo de datos esta ordenado por un atributo que no es llave, y este atributo es la llave de búsqueda para el árbol-B$^+$. Para cada valor de la llave K que aparece en el archivo de datos hay un par de llaves-punteros en un nodo hoja. El puntero va al primer registro que tiene a K como su valor de llave.

Hay aplicaciones adicionales de los árboles-B$^+$ las cuales permiten multiples ocurrencias de la llave de búsqueda, recordemos que la Figura 2.10 muestra un ejemplo de un árbol-B$^+$ sin llaves repetidas, para ver como se vería un árbol-B$^+$ con llaves repetidas véase la Figura 2.11.

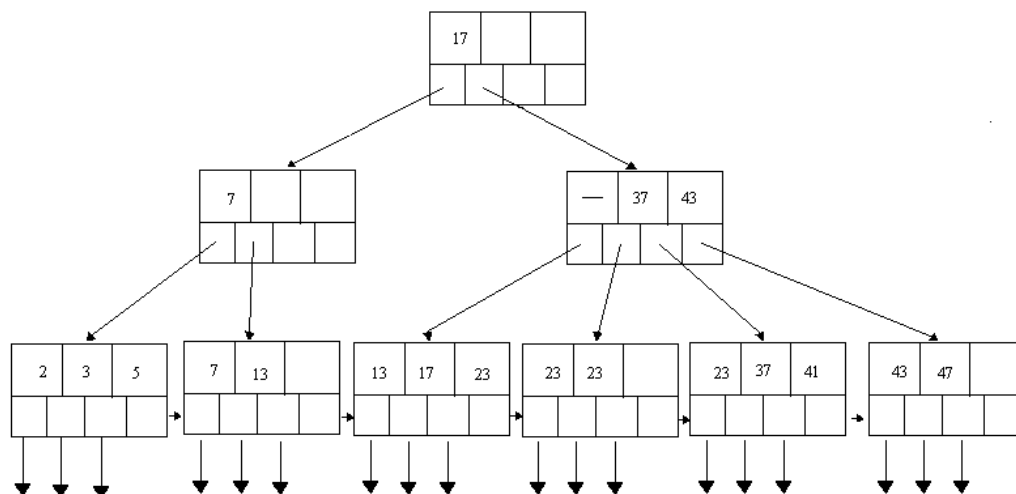

Figura 2.11: Ejemplo visual de un árbol-B$^+$ con llaves repetidas.

Vemos que la Figura 2.11 es muy similar a la Figura 2.10, pero con valores duplicados. Como podemos observar, la llave raíz es ahora 17 y no 13. La razón es que, aunque 13 es la llave más pequeña para el subárbol de la derecha, no es una llave única para este subárbol, debido a que también aparece en el subárbol de la izquierda. También observamos que el nodo interior derecho, tiene la primera posición como valor nulo. La razón es que su segundo nodo hijo (la cuarta hoja de izquierda a derecha) no tiene llaves nuevas del todo, vemos que 13 esta en el tercer nodo hoja, el 17 esta en la raíz y 23 también se encuentra en la siguiente hoja. Entonces si buscáramos el valor 23 u otro valor más pequeño, desearíamos empezar desde su primer hijo (tercera hoja), donde encontraríamos el valor que estamos buscando (si fuera 17), o encontraríamos el primer valor que estuviéramos buscando (si fuera 23).

Una vez que hemos estudiado la forma en que funcionan los árboles-B$^+$, pasamos a analizar su eficiencia para la operación de búsqueda.

Búsqueda en árboles-B$^+$

Supondremos que no hay llaves duplicadas en las hojas. También supondremos que el árbol-B$^+$ es un índice denso, entonces cada llave de búsqueda que aparece en el archivo de datos también aparecerá en un nodo hoja. Esto hará nuestra discusión sobre esta operación más simple.

Supongamos que tenemos un índice en forma de árbol-B$^+$ y queremos encontrar un registro con llave de búsqueda K. Buscamos K recursivamente, empezando desde la raíz y terminando en un nodo hoja. El procedimiento de búsqueda es el siguiente:

- **Base**: Si estamos en un nodo hoja, observar entre las mismas llaves. Si la i-esima llave es K, entonces el i-esimo puntero nos llevará al registro deseado.
- **Inducción**: Si estamos en un nodo interior con llaves $K_1, K_2, ..., K_n$, debemos seguir las reglas presentadas en la Sección 2.6 para determinar cual de sus hijos será examinado. Esto es, sólo hay un nodo hijo que nos puede llevar a un nodo hoja que contenga a la llave K. Si $K < K_1$, entonces éste corresponderá al primer hijo, si $K_1 \leq K < K_2$, entonces corresponderá al segundo hijo y así sucesivamente. Una vez situados en las hojas del árbol-B$^+$, se aplican los criterios del inciso anterior (Base).

Por otro lado, si deseamos encontrar todas la llaves en el rango [A,B] en las hojas de un árbol-B$^+$, tendríamos que hacer una búsqueda para encontrar la llave A. Este o no este, debemos dirigirnos a la hoja donde podría estar A, y luego buscar las hojas para las llaves que son mayores o iguales a A. Cada llave encontrada tendrá un puntero asociado a uno de los registros de datos cuyas llaves están en el rango deseado. Seguimos haciendo lo anterior hasta que encontremos una llave mayor que B.

Por último, los índices en árboles-B[5] son similares a los índices en árboles-B$^+$. La diferencia principal entre las dos estructuras es que un árbol-B elimina los valores redundantes de las llaves de búsqueda. En contraste con los árboles-B$^+$, donde cada valor de la llave de búsqueda aparece en algún nodo hoja.

[5] Si se desea estudiar más a profundidad este tipo de estructuras véase [2].

Para concluir el presente capítulo resta aclarar que todas estas organizaciones de archivo y estructuras de acceso a los datos fueron revisadas dado que al optimizar las consultas y las funciones de agregación, éstas son empleadas en las distintas estrategias de optimización.

Capítulo 3

Optimización de consultas y funciones de agregación

Ahora pasamos a estudiar las distintas maneras de optimizar los planes de ejecución de consultas y funciones de agregación. Empezamos estudiando de manera general los algoritmos que realizan distintos tipos de operaciones, posteriormente estudiamos las dos alternativas usualmente utilizadas para optimizar consultas y funciones de agregación.

El término optimización no es del todo correcto, ya que en algunos casos, el plan de ejecución elegido no es la estrategia óptima (la más eficiente) sino tan sólo una estrategia razonablemente eficiente. En general, encontrar la mejor solución consume demasiado tiempo.

Existen dos técnicas principales para optimizar las consultas:

1. *Basada en reglas heurísticas para ordenar las operaciones* en una estrategia de ejecución de una consulta. Por lo regular, las reglas reordenan las operaciones en un árbol de consulta y determinan un orden para ejecutar las operaciones.

2. *Basada en la estimación sistemática del costo* de diferentes estrategias de ejecución y elegir el plan con el más bajo costo estimado.

Empezamos nuestro estudio con los algoritmos principales para realizar las operaciones. Posteriormente estudiaremos el procesamiento de las consultas y donde entra la teoría de la optimización en este proceso, asimismo, profundizamos en las dos técnicas arriba mencionadas para optimizar consultas y funciones de agregación, posteriormente estudiaremos las dos alternativas de ejecutar las operaciones, y finalmente discutimos la elección de un plan de ejecución óptimo. Todo esto a efecto de mostrar el ciclo que toma una consulta y una función de agregación al momento de ser procesadas y optimizadas.

Cabe mencionar que a lo largo del capítulo se expondrá la optimización de consultas en general, y no sólo la optimización de las funciones de agregación, esto debido a que muchas de las técnicas que se utilizan para la optimización de consultas son igualmente utilizadas para las funciones de agregación. Sin embargo, para algunos casos nos detendremos a estudiar las funciones de agregación a efecto de dar una idea clara de cómo estas funciones instrumentan algunas técnicas de optimización[1].

3.1. Algoritmos de ejecución de consultas

En esta sección mostramos de manera general los algoritmos más utilizados para diversos tipos de operaciones[2].

Dividimos los algoritmos para realizar las operaciones dentro de tres categorías según su dificultad y su costo [17]:

- Métodos que involucran leer los datos de disco una sola vez. Estos son los *algoritmos de una pasada*. Frecuentemente funcionan sólo cuando al menos uno de los argumentos (relaciones) de la operación cabe en memoria principal.
- Métodos para volúmenes de datos que no caben en memoria principal. Este tipo de algoritmos llamados *algoritmos de dos pasadas* se caracterizan por leer los datos una primera vez del disco, se procesan de alguna

[1] Asimismo, se desea explicar las técnicas de optimización de las operaciones que se instrumentaron en conjunto con las funciones de agregación en el capítulo 5 (ejem. JOIN).
[2] Si desea ver un estudio más profundo sobre los algoritmos aquí presentados véase [17].

forma y se escriben nuevamente en disco, para posteriormente ser leídos una segunda ocasión para un procesamiento futuro durante una segunda pasada. Estos métodos se utilizan para el calculo de funciones de agregación en donde se requiera un agrupamiento.

- Métodos que funcionan sin un limite sobre el tamaño de los datos. Estos métodos utilizan tres o más pasadas, aunque éstos son generalizaciones recursivas de los algoritmos de dos pasadas.

En este capítulo nos concentraremos en los algoritmos de una y dos pasadas, los cuales utilizaremos para calcular las operaciones respectivas. Estas operaciones las clasificaremos según [17] en tres grupos:

- *Una tupla a la vez (operaciones unitarias).* Operaciones que no requieren la relación completa en memoria principal. Por ejemplo: la selección (σ) y la proyección (π). Los algoritmos son de una pasada para este tipo de operaciones.
- *Relación completa (operaciones unitarias).* Operaciones que necesitan analizar toda la relación para devolver el resultado. Por ejemplo, la eliminación de duplicados (γ), el agrupamiento (δ) y las funciones de agregación ($\Im$). Los algoritmos para calcular estas operaciones pueden ser de una pasada si la relación cabe en memoria, en otro caso, utilizaremos algoritmos de dos pasadas.
- *Relación completa (operaciones binarias).* Operaciones que necesitan más de una relación. Por ejemplo: la unión, la intersección, la diferencia y la reunión (join). Los algoritmos pueden ser de una pasada si al menos una de las relaciones cabe en memoria, en otro caso, utilizaremos algoritmos de dos pasadas. En el presente estudio revisamos este tipo de operaciones dado que evaluamos el desempeño de funciones de agregación sobre relaciones obtenidas a partir de reuniones.

3.1.1. Algoritmos para la selección

Existen diversos algoritmos de búsqueda para la selección de registros en un archivo. Estos algoritmos reciben el nombre de *exploradores de archivos* (*file*

scans), porque exploran los registros de un archivo para buscar y recuperar los registros que satisfacen una condición de selección. Si los algoritmos de búsqueda involucran el uso de índices, el índice de búsqueda es llamado *explorador indexado* (*index scan*). A continuación mostramos algunos algoritmos para la operación de selección:

- **B1 Búsqueda lineal**. En la búsqueda lineal se explora cada bloque de archivo y se comprueban todos los registros para determinar si satisfacen o no la condición de selección. El costo general de búsqueda es $B(R)$, donde $B(R)$ es el número de bloques del archivo R. Aunque, en promedio se puede decir que el tiempo de búsqueda es $B(R)/2$.

- **B2 Búsqueda binaria**. Si el archivo esta ordenado según un atributo y la condición de selección es una comparación de igualdad en ese atributo, podemos utilizar una búsqueda binaria para localizar los registros que satisfacen la selección. En el peor de los casos, el número de bloques que deben ser examinados es $\lceil \log_2(B(R)) \rceil$. Donde *B(R)* es el número de bloques del archivo R.

Estos algoritmos mostrados son implementados sobre archivos que no tienen un índice definido. Las estructuras basadas en índices se denominan *rutas de acceso* (*access path*), ya que proporcionan un camino a través del cual se pueden localizar y acceder a los datos. Recordemos que los algoritmos de búsqueda que utilizan un índice reciben el nombre de *exploradores indexados*. Algunos de estos algoritmos son:

- **S1. Índice primario**. Para una condición de igualdad sobre un atributo llave con un índice primario, se puede utilizar el índice para recuperar el único registro (por ser un atributo llave) que satisface la condición de igualdad correspondiente.

- **S2. Índice agrupado para recuperar multiples registros**. Si la condición de selección involucra una condición de igualdad sobre un atributo no llave con un índice agrupado, se utilizará el índice para recuperar todos los registros que satisfagan la condición.

- **S3. Índice secundario**. Este método de búsqueda puede ser utilizado para recuperar un simple registro si el atributo indexado es una llave (tiene valores únicos) o recuperar multiples registros si el atributo indexado es no llave (tiene valores repetidos).

Un SMBD tendrá disponible muchos de los algoritmos discutidos arriba, y generalmente varios adicionales. El *optimizador de consultas*[3] debe escoger el algoritmo apropiado para ejecutar cada operación de selección en una consulta. Esta optimización utiliza fórmulas que estiman el costo para cada uno de los métodos disponibles (los cuales veremos más adelante).

3.1.2. Algoritmos para la reunión (join)

La operación *reunión*[4] es una de las operaciones más importantes dentro del paradigma de bases de datos relacionales, sin embargo, es una de las operaciones que consume más tiempo al momento de ser procesadas, por esto, estudiaremos algunos de los algoritmos más utilizados para el calculo de este operador.

- **Reunión por iteración anidada**. Para cada registro t en R (bucle externo), se recupera cada registro s de S (bucle interno) y se analiza si los dos registros satisfacen la condición de la reunión $t_A = s_B$. Esto es:

  for each registro t_A **in** R **do begin**
      for each registro s_B **in** S **do begin**
          comprobar que el par (t_A, s_B) satisface la condición de la reunión
          si se cumple, se añaden al resultado.
      end
  end

[3]El optimizador de consultas se encarga de producir un plan de ejecución eficiente.
[4]A lo largo del trabajo utilizaremos los términos reunión y join indistintamente.

Este tipo de algoritmo son de una pasada y media (dado que una relación se lee una vez mientras que la otra se leerá repetidamente). Su costo es $B(S) + \frac{B(S)B(R)}{M-1}$, donde *B(R)* y *B(S)* son el número de bloques en R y S, respectivamente, y M es el número de bloques que cabe en memoria principal[5].

- **Reunión por iteración sencilla (utilizando una estructura de acceso para recuperar los registros emparejados)**. Supongamos que el atributo B de la relación S cuenta con un índice, entonces se recupera cada registro t en R, uno a la vez (bucle simple), y luego utilizamos la estructura de acceso (índice) para recuperar directamente todos los emparejamientos de los registros s de S que satisfacen $s_B = t_A$.

- **Reunión por ordenación-mezcla**. Si los registros de R y S están físicamente ordenados por los atributos de la reunión A y B, respectivamente, podemos implementar la reunión en la forma más eficiente posible. Ambos archivos son explorados concurrentemente en el orden de los atributos de la reunión. Si los archivos no están ordenados, se tendrán que ordenar primero utilizando el algoritmo *merge-sort*. La idea fundamental es copiar los registros en la memoria principal (en orden) y los registros de cada archivo son explorados sólo una vez por cada emparejamiento con el otro archivo, a menos que ambos atributos A y B de la reunión sean no llaves. Este método es muy eficiente si ambos archivos están ya ordenados sobre los atributos de la reunión. Una sola pasada es realizada sobre cada archivo, de aquí que el número de bloques accedidos es igual a la suma del número de bloques en ambos archivos. Sin embargo, si uno de los dos archivos no esta ordenado, éste debe ser ordenado para instrumentar la operación de reunión. Si estimamos el costo de la ordenación mediante el algoritmo *merge-sort* será proporcional a $B \log_2 B$ bloques accedidos, donde B es el número de bloques. Si ambos archivos necesitan ser ordenados, el costo de implementar la reunión por ordenación-mezcla será $(B(R) + B(S) + B(R) \log_2 B(R) + B(S) \log_2 B(S))$. Este tipo de algoritmo es de dos pasadas.

- **Reunión mediante *hash***. En este método de reunión mediante *hash*

[5] Se recomienda que la relación más grande vaya en el **for** externo, mientras que la relación más pequeña en el **for** interno, esto a efecto de tener un tiempo de ejecución menor.

se utiliza una función *hash h* para dividir las tuplas en ambas relaciones. La idea fundamental es dividir las tuplas de cada relación en conjuntos con el mismo valor dado por la función *hash*[6]. Para este método sólo una pasada es realizada sobre cada archivo, estén o no ordenados, esto a efecto de crear y guardar los bloques de registros en cubetas designadas por la función *hash*, una vez hecho esto, estas cubetas se escriben en disco, y finalmente se lee cada par de cubetas con el mismo valor *hash* a memoria principal para realizar un algoritmo de reunión de una pasada y poder obtener el resultado final. Como podemos observar este tipo de algoritmo es de dos pasadas, siendo su costo 3(B(R)+B(S)) E/S a disco.

3.2. Implementación de las funciones de agregación

Las funciones de agregación (MIN(), MAX(), CUENTA(), PROMEDIO(), SUMA(), CORRELACION(), etc), cuando se aplican a una tabla entera, se pueden calcular a través de un escaneo de la tabla[7] o al utilizar un índice apropiado, si éste esta disponible. Por ejemplo, consideremos la siguiente consulta en SQL:

SELECT MAX (salario)
FROM EMPLEADOS;

Si existe un índice en árbol-B^+ sobre el atributo *salario* para la relación *EMPLEADOS*, entonces el optimizador puede decidir utilizar el índice para buscar el valor más grande siguiendo el puntero más a la derecha de cada uno de los nodos del índice, desde la raíz hasta la hoja más a la derecha. Este nodo incluirá el *salario* más alto en su última entrada. En muchos casos, este escenario será más eficiente que hacer una exploración a toda la tabla. La función MIN(), puede ser implementada de una manera similar, excepto que el puntero más a la izquierda es seguido desde la raíz hasta la hoja más

[6]Se utiliza la misma función *hash* para ambos atributos de la reunión.
[7]Siendo su complejidad de tipo lineal.

a la izquierda. Ese nodo incluirá el *salario* más pequeño como su primera entrada.

Este índice podría ser utilizado también para las funciones de agregación: CUENTA(), PROMEDIO(), SUMA() entre otras. Pero sólo si es un índice denso, recordemos que esto significa que hay una entrada por cada registro en el archivo de datos. En este caso, el calculo asociado sería aplicado a los valores en el índice.

Cuando una cláusula GROUP BY es utilizada en una consulta, el operador de agregación debe ser aplicado separadamente para cada grupo de tuplas. Debido a esto, la tabla debe ser particionada dentro de subconjuntos de tuplas, donde cada grupo tiene el mismo valor para los atributos de agrupación.

Notemos que si existe un índice agrupado sobre el atributo de agrupamiento, los registros ya estarán agrupados dentro de los subconjuntos apropiados evitando así, una perdida de tiempo en su ordenamiento.

3.3. Procesamiento de las consultas

El *procesamiento de las consultas* consta de varios pasos, los cuales se muestran en la Figura 3.1.

La primera fase del procesamiento de las consultas es su traducción a su formato interno[8]. Este proceso de traducción es similar al trabajo que realiza el analizador de un compilador. Durante la generación del formato interno de una consulta, el analizador (*Parser*) comprueba la sintaxis de la consulta del usuario, verifica que los nombres de las relaciones que aparecen en ella sean nombres de relaciones en la base de datos, etc. Posteriormente se transforma la consulta a su equivalente en *álgebra relacional*, luego se construye un *árbol de consulta* a partir de las expresiones en álgebra relacional generadas por la consulta. En esta estructura de árbol es donde se aplican las técnicas de optimización como las reglas heurísticas del álgebra relacional y algunas

[8]En este trabajo sólo estudiaremos a partir de la fase dos, debido a que son las fases más relevantes para nuestro estudio. Entonces para todos los casos asumiremos que las consultas son correctas sintácticamente.

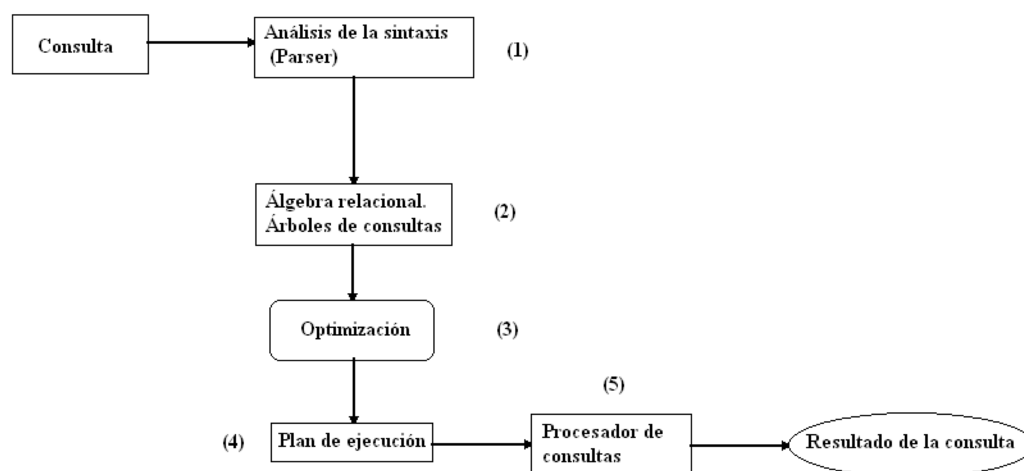

Figura 3.1: Ciclo del procesamiento de consultas.

técnicas para estimar el costo de las expresiones en el árbol de consulta. Esto debido a que dada una consulta existen varias maneras de obtener el mismo resultado [1].

Para especificar completamente como ejecutar una consulta, no basta con proporcionar la expresión del álgebra relacional, sino además hay que anotar en ella las instrucciones que especifiquen cómo ejecutar cada operación. Estas anotaciones podrían ser: el algoritmo a utilizar o el uso de algún índice (en caso de que exista alguno), entre otras. Las operaciones del álgebra relacional anotadas con instrucciones sobre su ejecución reciben el nombre de *primitivas de ejecución.* La secuencia de operaciones primitivas que se pueden utilizar en la ejecución de una consulta establece un *plan de ejecución de la consulta* (para ver un ejemplo véase la Figura 3.2). El *procesador de la consulta* escoge un plan de ejecución óptimo, lo ejecuta y devuelve la respuesta a la consulta.

Los diferentes planes de ejecución de una misma consulta pueden tener costos distintos. No podemos esperar que los usuarios escriban las consultas de manera que sugieran el plan de ejecución más eficiente. En su lugar, es responsabilidad del sistema construir un plan de ejecución de la consulta que minimice el costo de ejecución de la misma, a esta tarea la denominamos *optimización de consultas.*

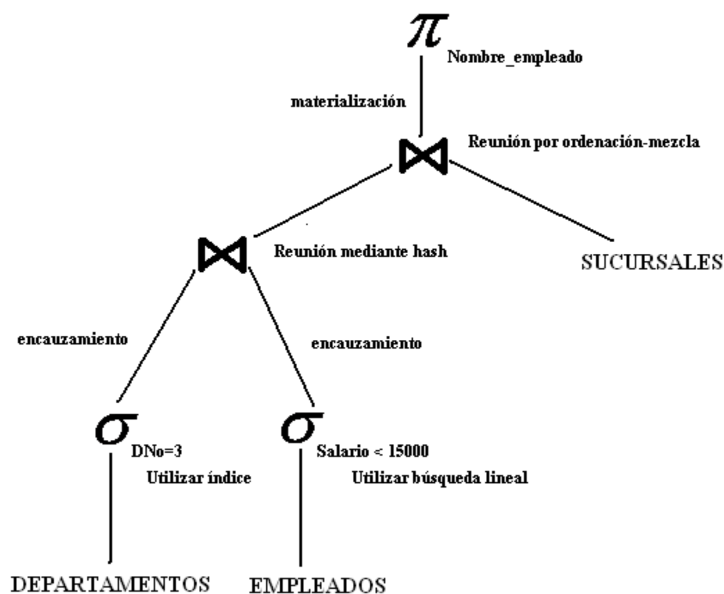

Figura 3.2: Un plan de ejecución.

Para optimizar una consulta, el optimizador de consultas debe conocer el costo de cada operación. Aunque el costo exacto es una tarea imposible de calcular, dado que depende de muchos parámetros como la memoria principal disponible, sin embargo, es posible obtener estimaciones aproximadas del costo de cada ejecución para cada operación.

Transformaciones de consultas

Hasta el momento dimos un panorama general del procesamiento de las consultas, en lo que resta del capítulo profundizaremos en algunas fases para finalmente obtener un plan de ejecución eficiente. Inicialmente una consulta en SQL es transformada a una expresión equivalente en álgebra relacional[9], para posteriormente transformar esta expresión en una estructura de árbol, denominada *árbol de consulta.*

[9]No profundizaremos en la transformación de consultas en SQL a una expresión en álgebra relacional, ni nos detendremos en explicar la sintaxis, daremos por hecho que el lector tiene conocimiento previo de esto. Para algunas referencias véase [6, 11].

Un *árbol de consulta* es una estructura en árbol que corresponde a una expresión del algebra relacional. Éste representa a las relaciones entrantes de la consulta como nodos hoja del árbol, y representa a las operaciones del álgebra relacional como nodos internos. Una ejecución del árbol de consulta consiste en ejecutar una operación de algún nodo interno y remplazar ese nodo interno por la relación resultante de aplicar esa operación. La ejecución termina cuando el nodo raíz es ejecutado y produce el resultado para la consulta [11].

Al inicio se genera un *árbol de consulta canónico*, esto es, un árbol de consulta sin ninguna optimización, como podemos notar, tal representación es extremadamente ineficiente si es ejecutada directamente. Es aquí donde entra el trabajo del *optimizador*, el cual transformará este árbol de consulta inicial mediante heurísticas de optimización a un árbol de consulta final que será más eficiente.

El optimizador debe incluir reglas de equivalencia entre las expresiones del álgebra relacional que puedan ser aplicadas a un árbol inicial. Las reglas heurísticas de optimización de consultas utilizan esta equivalencia para transformar el árbol inicial a un árbol de consulta ya optimizado. En la siguiente sección se presentan algunas de estas reglas heurísticas.

Con esto terminamos la fase de transformación de consultas (véase la Figura 3.2 para un ejemplo de un árbol de consulta) en SQL a un árbol de consulta, todas las consultas siguen el mismo procedimiento que mostramos en este apartado.

3.4. Heurísticas de optimización del álgebra relacional

En esta sección discutiremos técnicas de optimización que aplican reglas heurísticas para modificar la representación interna de una consulta y mejorar su desempeño.

Las consultas como sabemos se pueden representar de diferentes maneras con costos de ejecución diferentes. Diremos que dos expresiones del álgebra

relacional son *equivalentes* si en cada expresión se genera el mismo conjunto de tuplas [11]. El orden de las tuplas resulta irrelevante; puede que las dos expresiones generen las tuplas en un orden diferente, pero se considerarán equivalentes siempre que el conjunto de tuplas sea el mismo.

A continuación se muestran algunas reglas generales de equivalencia para las expresiones del álgebra relacional. Se utilizan $\theta_1, \theta_2, ..., \theta_n$ para denotar las condiciones respectivas, $A_1, A_2, ..., A_m$ como los atributos, $L_1, L_2, ..., L_i$ representan listas de atributos y por último R, S y T representan relaciones.

1. **Cascada de σ.** Las operaciones de selección conjuntiva pueden dividirse en una secuencia de selecciones individuales:

   $$\sigma_{\theta_1 AND \theta_2 AND ... AND \theta_n}(R) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(...(\sigma_{\theta_n}(R))...))$$

   Vemos que esta regla permite un mayor grado de libertad de movimiento de la operación de selección al permitirle descender entre las ramas del árbol.

2. **Conmutatividad de σ.** Las operaciones de selección son conmutativas:

   $$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

3. **Cascada de π.** Sólo son necesarias las últimas operaciones de una secuencia de operaciones de proyección, las demás pueden ser omitidas:

   $$\pi_{L_1}(\pi_{L_2}(...(\pi_{L_n}(R))...)) \equiv \pi_{L_1}(R)$$

4. **Conmutatividad de σ con π.** Si la condición de selección θ sólo involucra a los atributos $A_1, A_2, .., A_m$ en la lista de la proyección, entonces las dos operaciones pueden ser conmutadas:

   $$\pi_{A_1, A_2, ..., A_m}(\sigma_\theta(R)) \equiv \sigma_\theta(\pi_{A_1, A_2, ..., A_m}(R))$$

5. **Conmutatividad de $\bowtie$ (y $\times$).** La operación de reunión ($\bowtie$) es conmutativa, al igual que el producto cartesiano ($\times$):

   $R \bowtie_\theta S \equiv S \bowtie_\theta R$
   $R \times_\theta S \equiv S \times_\theta R$

   Notemos que aunque el orden de los atributos puede no ser el mismo en la relación resultante, el significado es el mismo.

6. **Distributividad de σ con $\bowtie$ (o $\times$).** Si todos los atributos en la condición de selección θ involucran sólo los atributos de una de las relaciones que van a ser reunidas, digamos R, las dos operaciones pueden ser distribuidas de la siguiente manera:

   $\sigma_\theta(R \bowtie S) \equiv (\sigma_\theta(R)) \bowtie S$

   Alternativamente, si la condición de selección θ puede ser escrita como (θ_1 AND θ_2), donde la condición θ_1 involucra sólo los atributos de R y la condición θ_2 involucra sólo los atributos de S, entonces podemos distribuirlas de la manera siguiente:

   $\sigma_\theta(R \bowtie S) \equiv (\sigma_{\theta_1}(R)) \bowtie (\sigma_{\theta_2}(S))$

   Las mismas reglas aplican si $\bowtie$ es reemplazado por la operación $\times$.

7. **Distributividad de π con $\bowtie$ (o $\times$).** Supongamos que la lista de proyecciones es de la forma $L = \{A_1, A_2, ..., A_n, B_1, B_2, ..., B_m\}$, donde $A_1, A_2, ..., A_n$ son atributos de R y $B_1, B_2, ..., B_m$ son atributos de S. Si la condición de la reunión θ involucra sólo atributos de L, entonces las dos operaciones pueden ser distribuidas de la siguiente manera:

   $\pi_L(R \bowtie_\theta S) \equiv (\pi_{A_1,A_2,...,A_n}(R)) \bowtie_\theta (\pi_{B_1,B_2,...,B_m}(S))$

8. **Asociatividad en $\bowtie$, $\times$, $\cap$ y $\cup$.** Estas cuatro operaciones son individualmente asociativas, esto es, si entendemos a Θ como cualquiera de las cuatro operaciones mencionadas, entonces tenemos:

$(R\Theta S)\Theta T \equiv R\Theta(S\Theta T)$

9. **Convirtiendo una secuencia de (σ, $\times$) a un $\bowtie$.** Las selecciones pueden combinarse con los productos cartesianos:

   $(\sigma_\theta(R \times S)) \equiv (R \bowtie_\theta S)$
   $(\sigma_{\theta_1}(R \bowtie_{\theta_2} S)) \equiv (R \bowtie_{\theta_1 AND \theta 2} S)$

A continuación mostramos otras heurísticas básicas para la optimización. Por ejemplo, otra heurística es aplicar primero las operaciones que reducen el tamaño del resultado. Esto es, aplicar tan pronto como sea posible las operaciones de selección para reducir el número de tuplas y la proyección para reducir el número de atributos. Esto se hace moviendo las operaciones de selección y de proyección tan abajo del árbol de consulta como sea posible. Asimismo, las operaciones de selección y reunión, las cuales son más restrictivas (devuelven una relación con menos tuplas), deberían ser ejecutadas antes de otra operación similar. Esto se realiza al reordenar los nodos hoja del árbol de consulta entre ellos mismos, mientras se evitan los productos cartesianos y se ajusta el resto del árbol apropiadamente.

También es importante obtener una buena ordenación en las reuniones para reducir el tamaño de los resultados temporales, por esto, la mayoría de los optimizadores de consultas prestan mucha atención en el orden de las reuniones. Como ya sabemos, las reunión es una operación conmutativa y asociativa, pero aunque se produzcan operaciones equivalentes, los costos de calcular cada una de ellas pueden ser diferentes. Entonces primero debemos asociar (o conmutar) aquellas reuniones que nos devuelvan la relación temporal más pequeña, para luego poder reunirla con las relaciones restantes siguiendo la misma lógica.

3.5. Estimación de los tamaños de las operaciones

Un optimizador de consultas no sólo debe depender de las reglas heurísticas, sino además, debería estimar y comparar los costos de ejecutar una consulta al implementar diferentes estrategias de ejecución y así poder escoger la estrategia con el menor *costo estimado*. Para que esto funcione, las estimaciones en el costo deben ser lo más precisas posible con el fin de que las distintas alternativas sean comparadas eficientemente. Asimismo, debemos limitar el número de estrategias de ejecución, en otro caso, perderíamos mucho tiempo haciendo estimaciones de costo para muchas estrategias de ejecución.

Llamaremos a esta aproximación *optimización de consultas basada en costos*. Ésta utiliza técnicas de optimización tradicionales que buscan dentro del *espacio de soluciones* de un problema una solución que minimice la *función de costo*. Las funciones de costo utilizadas en la optimización de consultas son estimaciones, por tanto, no son exactas. Entonces podríamos escoger una estrategia de ejecución que no sea la más optima.

A continuación estudiamos algunas de las estadísticas necesarias para poder calcular las funciones de costo, estas estadísticas están almacenadas en el catálogo del sistema. Por lo tanto, echaremos un vistazo a la información incluida en éste.

Información del catálogo

Los catálogos de los SMBDs almacenan las siguientes estadísticas sobre las relaciones de las bases de datos, de donde son accedidas por el optimizador de consultas:

- **T(R)**, representa el número de tuplas de la relación R.
- **B(R)**, representa el número de bloques necesarios para mantener a todas las tuplas de la relación R.
- **S(R)**, representa el tamaño de cada tupla de la relación R en bytes.

- **F(R)**, representa el factor de bloqueo para la relación R, es decir, el número de tuplas de la relación R que caben en un bloque.
- **V(R,A)**, representa el número de valores distintos que aparecen en la relación R para el atributo A. Este valor será igual a $\pi_A(R)$. Si A es una llave de la relación R.

Asimismo, se conservan dentro del catálogo la información de los índices, como las alturas de los árboles-B^+. Por ejemplo, en algunas funciones de costo el número de bloques del primer nivel del índice es necesario.

La información del número de niveles de los índices es fácil de mantener debido a que no cambian muy a menudo. Sin embargo, otra información puede cambiar frecuentemente. Por tanto, si deseamos conservar estadísticas precisas, cada vez que se modifica una relación también hay que actualizar las estadísticas, esta actualización supone una sobrecarga sustancial por lo que la mayoría de los sistemas no actualizan las estadísticas con cada modificación. En lugar de esto, actualizan su información durante períodos de poca carga en el sistema, en consecuencia, puede que las estadísticas utilizadas para escoger una estrategia de procesamiento de consultas no sean completamente exactas, sin embargo, aún cuando no se produzcan demasiadas actualizaciones de las estadísticas, éstas serán lo bastante precisas como para proporcionar una buena estimación de los costos relativos de los diferentes planes [1].

La información aquí presentada está simplificada. Los optimizadores reales suelen conservar información estadística adicional para mejorar la precisión de los planes de ejecución.

A continuación se utilizan las estadísticas proporcionadas por el catálogo del sistema para poder calcular las siguientes funciones de costo.

Estimación del tamaño de la Selección

Cuando ejecutamos una selección, generalmente reducimos el número de tuplas, aunque el tamaño de las tuplas permanezca igual. Sólo estudiaremos dos tipos de selección (para dar una idea general), donde un atributo es igualado a una constante y cuando existe una comparación de desigualdad. Para el

primer caso, existe una manera fácil de estimar el tamaño del resultado, sea $S = \sigma_{A=c}(R)$, donde A es un atributo de R y c es una constante. Entonces una posible estimación de la función de costo para la selección de este tipo es:

$$T(S) = T(R)/V(R, A)$$

Esta regla se mantiene si todos los valores para el atributo A tienen la misma frecuencia de ocurrencia en la base de datos, sin embargo, es la mejor estimación aún cuando este supuesto no se cumpla.

El tamaño estimado es más complicado cuando la selección involucra una comparación de desigualdad, por ejemplo, $S = \sigma_{A<c}(R)$. Podríamos pensar que en promedio, la mitad de las tuplas satisface la comparación y la otra mitad no, entonces $T(R)/2$ sería el tamaño estimado para S. Sin embargo, hay una intuición de que las consultas que involucran una desigualdad tienden a recuperar una pequeña fracción de las tuplas posibles. Entonces, se suele proponer una aproximación que asume que la desigualdad recuperará cerca de un tercio de las tuplas, en lugar de la mitad de ellas. Por lo tanto, si $S = \sigma_{A<c}(R)$, entonces nuestra estimación para *T(S)* es:

$$T(S) = T(R)/3$$

Estimación del tamaño de la Reunión (Join)

Para analizar este tipo de operaciones sólo consideraremos el caso de la reunión natural (NATURAL JOIN). Únicamente estudiaremos la reunión natural donde se involucra la igualdad de dos atributos (a manera de ejemplo). Esto es, estudiaremos reuniones de la forma $R(X, Y) \bowtie S(Y, Z)$, donde Y es una atributo simple, aunque X y Z puedan representar cualquier conjunto de atributos.

Existen varios problemas posibles con respecto a cómo se relacionan los valores de Y en R y S. Por ejemplo:

1. Las dos relaciones podrían tener conjuntos disjuntos en los valores de Y, en cuyo caso la reunión será vacía y $T(R \bowtie S) = 0$.

2. Si Y es una llave de S y una llave foránea de R, entonces cada tupla de R se reúne con exactamente una tupla de S, y $T(R \bowtie S) = T(R)$.

3. Cuando casi todas las tuplas de R y S tienen el mismo valor en Y, $T(R \bowtie S)$ será cercano a $T(R)T(S)$.

A continuación trataremos los casos más comunes de reunión, pero antes haremos la siguiente asunción.

Contención de conjuntos de valores. Si Y es un atributo que aparece en varias relaciones, entonces cada relación elige varios de los posibles valores $(y_1, y_2, ...)$. Como consecuencia, si R y S son dos relaciones con un atributo Y, y $V(R,Y) \leq V(S,Y)$, entonces cada valor Y será un valor de S. Este enfoque se presenta (entre otros) cuando Y es una llave de S y llave foránea de R [17].

Bajo este supuesto, podemos estimar el tamaño para $R(X,Y) \bowtie S(Y,Z)$ como sigue. Sea $V(R,Y) \leq V(S,Y)$. Entonces cada tupla t de R tiene un probabilidad de $1/V(S,Y)$ de reunirse con una tupla de S. Sabemos que *T(S)* es el número de tuplas en S, entonces el número esperado de tuplas reunidas es $T(S)/V(S,Y)$. Además como hay *T(R)* tuplas de R, el tamaño estimado de $R \bowtie S$ es $T(R)T(S)/V(S,Y)$. Por otra parte, si $R(X,Y) \geq S(Y,Z)$, entonces la estimación es $T(R \bowtie S) = T(R)T(S)/V(R,Y)$, en general, dividimos por la que resulte más grande entre $V(S,Y)$ y $V(R,Y)$. Quedando finalmente la función de costo como:

$$T(R \bowtie S) = T(R)T(S)/max\{V(R,Y), V(S,Y)\}$$

En general, la siguiente regla es utilizada para estimar el tamaño de una reunión natural cuando hay cualquier cantidad de atributos compartidos entre las dos relaciones.

La estimación del tamaño de $R \bowtie S$ *es calculado al multiplicar* $T(R)$ *por* $T(S)$ *y dividido por el mas grande de entre* $V(R,y)$ *y* $V(S,y)$ *para cada atributo* y *que es común para* R *y* S.

Estimación del tamaño de otras operaciones

Hemos visto dos operaciones (selección y reunión) con fórmulas exactas en el número de tuplas dentro del resultado, sin embargo, para las operaciones restantes el tamaño del resultado no es tan fácil de determinar. Revisaremos otras operaciones del álgebra relacional y daremos algunas aproximaciones de como estimarlas.

- **Unión**, una unión puede ser tan grande como la suma de los tamaños o tan pequeña como la más grande de las dos relaciones. Se sugiere la suma de la relación más grande más la mitad de la más pequeña.
- **Intersección**, el resultado puede tener desde cero tuplas hasta la misma cantidad de tuplas como la relación más pequeña, una aproximación es tomar el promedio, esto es, la mitad de la relación más pequeña.
- **Diferencia**, cuando calculemos $R - S$, el resultado puede tener entre $T(R)$ y $T(R) - T(S)$ tuplas. Se sugiere tomar el promedio como una estimación: $T(R) - \frac{1}{2}T(S)$.
- **Agregación**, el tamaño de ${}_{A_i}\Im_L(R)$, donde A_i son los atributos de agrupamiento y L es la lista de funciones de agregación, su estimación será simplemente $V(R, A_i)$, debido a que el número de tuplas de ${}_{A_i}\Im_L(R)$ es igual al número de grupos.

3.6. Ejecución de expresiones

La manera ingenua de ejecutar una expresión es simplemente ejecutar una operación a la vez en un orden apropiado. El resultado de cada ejecución se **materializa** en una relación temporal para su inmediata utilización. Un inconveniente de esta aproximación es la necesidad de construir las relaciones temporales, las cuales se tienen que escribir a disco. Un enfoque alternativo es ejecutar varias operaciones de manera simultánea en **encauzamiento** (*pipeline*), con los resultados de una operación pasados a la siguiente, sin necesidad de almacenar relaciones temporales.

En la **materialización** se ejecutan las operaciones utilizando alguno de los algoritmos estudiados en la Sección 3.1, almacenando sus resultados en relaciones temporales. Luego se utilizan estas relaciones temporales para ejecutar las operaciones del siguiente nivel del árbol de consulta (vistos en la Sección 3.3), cuyas entradas son las relaciones temporales o relaciones almacenadas en la base de datos. Repitiendo este proceso se calcularía finalmente la operación en la raíz del árbol, y así obtendremos el resultado final de la expresión.

Por otra parte, podemos mejorar la eficiencia de la ejecución de consultas mediante la reducción del número de archivos temporales que se producen. Se lleva a cabo esta reducción con la combinación de varias operaciones relacionales en forma de **encauzamiento**, en el que se pasan los resultados de una operación a la siguiente operación del encauzamiento. La combinación de operaciones en un encauzamiento elimina el costo de leer y escribir las relaciones temporales. La selección y la proyección son excelentes candidatas para el encauzamiento, debido que estas operaciones son de una tupla a la vez (descritos en la Sección 3.1). Sin embargo, hay casos donde la materialización es la única alternativa para realizar la operación (ejem. Agrupamiento).

3.7. Elección del plan de ejecución

La generación de expresiones sólo es una parte del proceso de optimización de consultas, ya que cada operación de la expresión puede implementarse con algoritmos diferentes. Por tanto, se necesita un plan de ejecución para definir exactamente el algoritmo que se utilizará para cada operación y el modo en cómo se coordinará la ejecución de las operaciones. Como vimos en la Sección 3.1, se pueden emplear diferentes algoritmos para cada operación relacional, lo que da lugar a planes de ejecución alternativos.

Una manera de escoger un plan de ejecución para una expresión de consulta es sencillamente escoger el algoritmo más económico para ejecutar cada operación. Se puede escoger cualquier ordenación de las operaciones que asegure que las operaciones ubicadas por debajo en el árbol de consulta se ejecuten antes que las operaciones situadas más arriba. Sin embargo, la selección del algoritmo más económico para cada operación no es necesariamente la mejor opción. Por ejemplo, puede ser que una reunión por ordenación-mezcla en

un nivel del árbol resulte más costosa que una reunión basada en *hash*, pero proporcione un resultado ordenado que haga más económica la ejecución de operaciones posteriores. Entonces al escoger el mejor algoritmo global hay que considerar incluso los algoritmos no óptimos para cada una de las operaciones. Por lo tanto, además de considerar las expresiones alternativas de cada consulta, también hay que considerar los algoritmos alternativos para cada operación de cada expresión. Para así obtener un plan de ejecución óptimo[10] [1].

Dado un plan de ejecución, se puede estimar su costo empleando las estadísticas estimadas en la Sección 3.5 junto con las estimaciones dadas en la Sección 3.1 para los algoritmos. Por otro lado, dependiendo de los índices disponibles, ciertas operaciones de selección y agregación se pueden ejecutar sólo con un índice y sin acceder a la relación en sí. Sin embargo, esto nos sigue dejando con el problema de la selección del mejor plan de ejecución para la consulta.

Algunos optimizadores se basan en el costo para seleccionar el plan de ejecución más óptimo, esto es, generan una gama de planes de ejecución a partir de la consulta dada, empleando las reglas heurísticas (vistas en las Sección 3.4) y escogen el plan con el costo mínimo. Para consultas complejas el número de planes de ejecución diferentes puede ser grande, entonces se suele utilizar un *algoritmo de programación dinámica*[11] para encontrar el ordenamiento y algoritmos óptimos (por ejemplo, para encontrar el mejor orden para la reunión). Los algoritmos de programación dinámica almacenan los resultados calculados y los vuelven a utilizar, un procedimiento que puede reducir enormemente el tiempo de ejecución.

Sin embargo, un inconveniente de optimizar de esta manera es el costo de la propia optimización. Debido que aunque el costo de la optimización de las consultas se puede reducir mediante algoritmos inteligentes, el número de planes de ejecución distintos para una consulta puede ser muy grande y buscar el plan óptimo a partir de este conjunto requiere un gran esfuerzo de cómputo. Por ello, muchos sistemas utilizan algunas heurísticas para reducir el costo de la optimización (al reducir el espacio de búsqueda). Una de las heurísticas es la siguiente regla para la transformación de consultas del álge-

[10] Recordemos que muchas de estas técnicas sólo son utilizadas por los optimizadores de consultas de los SMBDs.

[11] Si se desea saber más sobre este tipo de algoritmos, véase [1]

bra relacional: "*Realizar las operaciones de selección tan pronto como sea posible*". Los optimizadores utilizan esta regla sin averiguar si se reduce el costo mediante esta transformación.

La operación de proyección, asimismo, reduce el tamaño de las relaciones. Por tanto, siempre que haya que generar una relación temporal, resulta ventajoso aplicar inmediatamente cuantas proyecciones sea posible. Entonces otra heurística es: "*Realizar las proyecciones tan pronto como sea posible*". Suele resultar mejor llevar acabo las selecciones antes que las proyecciones, ya que las selecciones tienen la posibilidad de reducir mucho el tamaño de las relaciones y permiten el empleo de índices para tener acceso a las tuplas.

Los optimizadores de consultas tienen más heurísticas para reducir el costo de la optimización. Por ejemplo, muchos optimizadores no toman en consideración todos los posibles ordenamientos para la reunión, sino que restringen la búsqueda a tipos concretos de ordenación de la reunión, un ejemplo sería ejecutar las reuniones de izquierda a derecha.

Resumiendo, podemos observar que el objetivo principal de la optimización es obtener un plan óptimo para ejecutar una consulta o función de agregación dada, esta optimización es realizada a través de todos los métodos aquí expuestos como: los algoritmos de las operaciones, las rutas de acceso existentes, reglas heurísticas del álgebra relacional, estimaciones del tamaño de las operaciones, formas de ejecutar las expresiones (materialización o encauzamiento) y heurísticas para reducir el espacio de búsqueda. Todas estas técnicas (y muy probablemente otras más) son combinadas de tal manera que produzcan el plan de ejecución con el menor costo posible. Una vez que el SMBD a encontrado un plan de ejecución óptimo éste lo ejecuta para finalmente devolver el resultado final.

Capítulo 4

Funciones Definidas por el Usuario (UDFs)

Ahora pasamos a estudiar la parte central de nuestro trabajo. Le dedicamos el presente capítulo al estudio de las UDFs debido a que de acuerdo a nuestros resultados experimentales la instrumentación via UDFs resultó ser uno de los mecanismos con mejor desempeño[1].

Una UDF es una subrutina desarrollada en ciertos lenguajes de programación[2], para este trabajo fueron: C para PostgreSQL y MySQL y *C#* para SQL Server 2005. Estas UDFs son compiladas[3] a código objeto (PostgreSQL y MySQL) y a dll (SQL Server), posteriormente son enlazadas con los respectivos SMBDs, finalmente pueden ser utilizadas mediante una sentencia "SELECT", como cualquier función en SQL. Las UDFs representan un API que permite a un usuario final extender las funcionalidades de un SMBD [27] y además permiten ejecutar más eficientemente algunos tipos de

[1] Cabe mencionar que no existe actualmente un documento en donde se haga un estudio comparativo de funciones de agregación en distintos SMBDs.

[2] Esto dependerá del lenguaje soportado por el SMBD.

[3] Recordemos que las funciones en SQL son interpretadas, esto es uno de los puntos por lo que las UDFs resultaron ser más eficientes. Pero los diseñadores de SMBDs prefirieron perder un poco de eficiencia y ganar mayor flexibilidad al utilizar un lenguaje interpretado y no compilado.

funciones[22].

Las UDFs se dividen en dos tipos: (1) *UDFs escalares*, las cuales toman un número de parámetros y devuelven un solo valor por cada fila y (2) *UDFs agregadas*, éstas devuelven una fila por cada agrupamiento distinto. Para el caso donde no haya agrupación, entonces únicamente devuelven un solo valor.

En este capítulo estudiaremos primero las UDFs escalares y posteriormente abordaremos las UDFs agregadas para cada uno de los SMBDs antes mencionados.

4.1. MySQL

4.1.1. UDFs escalares

Iniciamos realizando un estudio detallado del manejo y construcción de las UDFs en MySQL. Posteriormente abordaremos la metodología para la construcción de las UDFs para los dos SMBDs restantes.

Denotaremos a `xxx()` como el nombre de una función ejemplo en C (UDF) y la función `XXX()` como el nombre de la función en SQL.

Las funciones en C que se deben escribir para implementar la interface para `XXX()` son:

- **xxx()**: La **función principal** es donde el resultado de la función se calcula. La correspondencia entre los tipos de datos de la función en SQL y el tipo de retorno de la función en C se muestran en el Cuadro 4.1:

- **xxx_init()**:

  Esta función es opcional, sirve para:

  1. Verificar el número de argumentos para `XXX()`.

Tipo en SQL	Tipo en C
STRING	char *
INTEGER	long long
REAL	double

Cuadro 4.1: Correspondencia entre tipos de datos en SQL y C en MySQL

2. Verificar que los argumentos sean de un determinado tipo, alternativamente puede forzar a los argumentos a ser de un tipo específico.
3. Reserva cualquier memoria requerida por la función principal.
4. Especifica la longitud máxima del resultado.
5. Especifica (para funciones numéricas) el máximo número de decimales de salida.
6. Especifica si el resultado puede ser NULL.

- **xxx_deinit()**: Esta función es opcional, libera cualquier memoria reservada por la función de inicialización (xxx_init()).

  Cuando un comando en SQL invoca a `XXX()`, se llama a la función de `xxx_init()` para realizar cualquier inicialización requerida, tales como la verificación de argumentos o la reservación de memoria. Si `xxx_init()` devuelve un error, el comando SQL se aborta con un mensaje de error y no se invocará la función principal ni la función de-inicialización (xxx_deinit()). En caso contrario, se llama a la función principal `xxx()` una vez para cada registro. Tras procesar todos los registros, se llama a la función de-inicialización `xxx_deinit()` para que pueda realizar cualquier limpieza requerida.

A continuación detallamos las propiedades de las funciones anteriores:

La **función principal** `xxx()` debe declararse como se muestra a continuación:

El tipo de retorno y los parámetros diferirán dependiendo de la declaración de la función en SQL (`XXX()`), dependiendo de que tipo sea utilizado dentro del

comando CREATE FUNCTION. A continuación mostramos las diferentes alternativas para la declaración de la función principal:

Para funciones STRING:

```
char *xxx(UDF_INIT *initid, UDF_ARGS *args, char *result,
            unsigned long *length, char *is_null, char *error);
```

Para funciones INTEGER:

```
long long xxx(UDF_INIT *initid, UDF_ARGS *args,
                             char *is_null, char *error);
```

Para funciones de tipo REAL:

```
double xxx(UDF_INIT *initid, UDF_ARGS *args,
                             char *is_null, char *error);
```

Las **funciones de inicialización y de-inicialización** se declaran de la siguiente manera:

```
my_bool xxx_init(UDF_INIT *initid,UDF_ARGS *args,char *message);
```

Y la función de-inicialización:

```
void xxx_deinit(UDF_INIT *initid);
```

El parámetro `initid` es definido en las tres funciones (principal, inicialización y de-inicialización). Este apunta a la estructura `UDF_INIT` que es utilizada para comunicar información entre las funciones. Los miembros de la estructura `UDF_INIT` se muestran a continuación.

my_bool maybe_null

En la función `xxx_init()` debemos asignar a *maybe_null* el valor de 1 si `xxx()` puede devolver NULL o cero en caso contrario. El valor por defecto es 1.

unsigned int decimals

Nos indica el número de decimales de salida. El valor por defecto es el número máximo de decimales en los argumentos pasados a la función principal. (Por ejemplo, si a la función se le pasan 1.34, 1.345, y 1.3, el valor por defecto es 3, ya que 1.345 tiene 3 decimales.

unsigned int max_length

Nos permite definir la longitud máxima del resultado. El valor por defecto `max_length` difiere en función del tipo de resultado de la función. Para funciones de cadenas de caracteres, el valor por defecto es el argumento más largo. Para funciones enteras, el valor por defecto es de 21 dígitos. Para funciones reales, el valor por defecto es 13 más el número de decimales indicados por `initid−` >`decimals`. (Para funciones numéricas, la longitud incluye cualquier signo o carácter de punto decimal.)

char ∗ptr

Es un puntero que la función puede utilizar para algún propósito. Por ejemplo, las funciones pueden usar initid− >ptr para comunicar memoria reservada entre ellos. xxx_init() debe reservar la memoria y asignarla al puntero:

$initid- > ptr = allocated_memory;$

En `xxx()` y `xxx_deinit()`, nos referiremos a `initid−` >`ptr` para utilizar o liberar la memoria.

El parámetro *args* apunta a una estructura `UDF_ARGS` que tiene los miembros listados a continuación:

unsigned int arg_count

Utilizamos este valor en la función de inicialización si necesitamos que la función sea llamada con un número particular de argumentos. Por ejemplo:

```
if (args->arg_count != 2)
{
    strcpy(message,‘‘XXX() requiere dos argumentos de entrada");
    return 1;
}
```

Para asegurar que los argumentos sean de un tipo y devuelva un error en caso contrario, se utiliza `arg_type` en la función de inicialización. Por ejemplo:

```
if (args->arg_type[0] != STRING_RESULT  ||
                             args->arg_type[1] != INT_RESULT)
{
    strcpy(message,‘‘XXX() requiere una cadena y un entero");
    return 1;
}
```

Compilar e instalar UDFs

El archivo que contiene nuestras UDFs debe compilarse e instalarse en el equipo donde corre el servidor. Las instrucciones para realizar lo anterior son[4]:

Los archivos.c en donde tenemos nuestro código correspondiente a nuestra UDF, debe compilarse como una biblioteca compartida (código objeto), supongamos que nuestro archivo se llama *udf_ejemplo.c* y queremos crear una biblioteca compartida (.so), llamada *udf_ejemplo.so*, esto se realiza de la siguiente manera:

shell > gcc   − shared   − o   udf_ejemplo.so   udf_ejemplo.c

Tras compilar una biblioteca compartida la cual contiene nuestras UDFs, debemos instalarla y registrarla en MySQL. Para lograr esto copiamos la biblioteca compartida *udf_ejemplo.so* en un directorio como: */usr/lib* en donde

[4]Recordemos que para MySQL y PostgreSQL las instrucciones de compilación e instalación serán para el sistema operativo Linux.

lo buscará el enlazador dinámico del sistema (en tiempo de ejecución). Cuando la biblioteca compartida se ha instalado, tenemos que notificar a MySQL las nuevas funciones con los siguientes comandos:

```
CREATE FUNCTION XXX() RETURNS Tipo_de_dato
SONAME  "udf\_ejemplo.so";
```

Las UDFs pueden borrarse con la sentencia DROP FUNCTION:

```
DROP FUNCTION  XXX();
```

Los comandos `CREATE FUNCTION` y `DROP FUNCTION` actualizan la tabla del sistema `func` en la base de datos `mysql`. El nombre de la función, tipo y nombre de la biblioteca compartida se almacenan en la tabla. Debemos tener los privilegios `INSERT` y `DELETE` para la base de datos `mysql` para crear y borrar funciones.

No debemos usar `CREATE FUNCTION` para añadir una función que se ha creado previamente. Si necesitamos reinstalar una función, debemos borrarla con `DROP FUNCTION` y volver a instalarla con `CREATE FUNCTION`. Podemos necesitar hacer esto, por ejemplo, si recompilamos una nueva versión de nuestra función, de forma que MySQL tenga la nueva versión. De otro modo, el servidor continua implementando la versión anterior.

Una función activa es aquella función que se ha cargado con `CREATE FUNCTION` y no se ha borrado con `DROP FUNCTION`. Todas las funciones activas se vuelven a cargar cada vez que el servidor inicia, a no ser que iniciemos MySQL con la opción $--skip-grant-tables$. En ese caso, la inicialización de UDFs no se hace y no estarán disponibles.

4.1.2. UDFs agregadas

Para funciones de agregación que funcionan como la función SUM(), debemos proporcionar además de las tres funciones explicadas anteriormente (principal, inicialización y de-inicialización) las dos funciones siguientes:

- `xxx_clear()`: Resetea el valor agregado actual, sin insertar el argumento como valor agregado inicial para un nuevo grupo.

- `xxx_add()`: Añade el argumento al valor agregado actual.

Todas las funciones deben ser flujos seguros. Esto incluye no sólo la función principal, sino también las funciones de inicialización o de-inicialización, y las funciones adicionales requeridas por las funciones agregadas. Una consecuencia de esta restricción es que no se le permite reservar ninguna variable global. Si necesitamos memoria, debemos reservarla en `xxx_init()` y liberarla mediante la función `xxx_deinit()`.

A continuación se explican a detalle estas funciones (xxx_clear y xxx_add):

`xxx_clear()`

Se invoca al principio para cada nuevo grupo, pero también puede invocarse para resetear los valores para una consulta donde no hayan registros que coincidan con la búsqueda. Esta función se declara como sigue:

```
char *xxx_clear(UDF_INIT *initid, char *is_null, char *error);
```

is_null se fija para apuntar a `CHAR(0)` antes de llamar a `xxx_clear()`. Si algo falla, puede almacenar un valor en la variable a la que apunta el argumento *error*. Este argumento apunta a una variable de un byte, no a un buffer cadena.

`xxx_add()`

Esta función es invocada para todos los registros que pertenecen al mismo grupo, excepto para el primer registro. Debemos utilizarla para añadir los valores en el argumento `UDF_ARGS` a su variable de agregación interna. Se declara de la siguiente manera:

```
char *xxx_add(UDF_INIT *initid, UDF_ARGS *args,
                                  char *is_null, char *error);
```

Por último la función `xxx()` para una UDF agregada debe declararse de la misma manera como lo hicimos anteriormente (UDFs escalares). Para una UDF agregada, MySQL llama a la función `xxx()` una vez que todos los registros en el grupo han sido procesados. *∗is_null* se resetea para cada grupo (antes de llamar `xxx_clear()`). ∗`error` nunca se resetea.

A continuación mostramos un resumen de como maneja MySQL las UDF agregadas:

1. Invoca la función **xxx_init()**, la cual asigna cualquier memoria que necesitemos para almacenar los resultados.
2. Ordena la tabla de acuerdo a la expresión GROUP BY.
3. Invoca a la función **xxx_clear()** para la primera fila de cada nuevo grupo.
4. Invoca a la función **xxx_add()** para cada nueva fila que pertenezca al mismo grupo.
5. Invoca a la función principal **xxx()** para obtener el resultado para la agregación cuando el grupo cambia o después de que la última fila ha sido procesada.
6. Repite los pasos 3-5 hasta que todas las filas han sido procesadas.
7. Invoca a la función **xxx_deinit()** para liberar cualquier memoria que ha sido asignada anteriormente por la función de inicialización.

Para observar un ejemplo completo de UDFs agregadas véase el apéndice B.

El registro de las funciones de agregación[5] es de forma similar a las UDFs escalares (no agregadas), pero ahora utilizamos el comando CREATE AGGREGATE para este fin, por ejemplo:

[5]La compilación e instalación de la biblioteca compartida es de la misma forma que para las UDFs escalares.

CREATE AGGREGATE FUNCTION correlation RETURNS REAL SONAME 'aggregations.so'

Supongamos que la biblioteca compartida es *aggregations.so*, en donde se encuentra el código de la función *correlation*[6]. Como podemos observar, no definimos ningún parámetro de entrada, esto fue hecho dentro del código de la UDF agregada (en la función de inicialización) en donde especificamos que sólo permitirá el ingreso de dos columnas de tipo real, en caso contrario, un error será devuelto. Asimismo, cuando queremos borrar una UDF agregada lo realizamos de la siguiente manera:

DROP AGGREGATE correlation;

Para observar la referencia completa de la forma de construir UDFs escalares y agregadas en MySQL véase [15].

4.2. PostgreSQL

4.2.1. UDFs escalares

Ahora pasamos hacer un estudio detallado de las UDFs en PostgreSQL.

Existen dos clases de funciones que pueden ser usadas en una sentencia "SELECT".

- **Funciones escalares**, las cuales toman un número de parámetros y devuelven un solo valor por cada fila.
- **Funciones de agregación**, estas devuelven una fila para cada agrupamiento distinto. Para el caso de no haber agrupación, entonces únicamente devuelven un solo valor (ejem. SUM ()).

PostgreSQL divide estas dos clases de funciones al momento de ser imple-

[6] Véase el apéndice B para observar el código completo.

mentadas como Funciones Definidas por el Usuario, las funciones escalares pueden ser construidas mediante UDFs, mientras que las funciones de agregación son implementadas a través de Agregaciones Definidas por el Usuario (UDAs, por sus siglas en ingles), ambas para nuestro estudio en lenguaje C, aunque nosotros las identificaremos como UDFs agregadas como en [25].

Empezaremos nuestro estudio con las funciones escalares, nuevamente el procedimiento general es similar al explicado en la Sección 4.1, las funciones son construidas en lenguaje C, estas funciones son compiladas a código objeto y posteriormente cargadas en el SMBD. Una vez realizado esto puede ser utilizada la UDF en una sentencia "SELECT", como cualquier función en SQL.

Existen dos formas distintas de implementar funciones en C en PostgreSQL: (1) La "versión 1" es la forma más nueva de implementar funciones en C, la cual utiliza ciertas macros incluidas en las bibliotecas de PostgreSQL, esto con el fin de hacer más simple su construcción. (2) La "versión 0" no implementa estas macros, y por lo mismo este viejo estilo dejo de ser aprobado, esto debido a problemas de portabilidad y limitaciones en funcionalidad. Por lo anterior sólo nos enfocaremos al estudio de la "versión 1".

Carga dinámica de funciones

La primera vez que una UDF es llamada en una sesión, el "cargador dinámico" carga el código objeto (o biblioteca compartida) a memoria para que la función pueda ser invocada. La sentencia CREATE FUNCTION para una UDF debe por lo tanto especificar dos cosas: el nombre del código objeto (ejem. funcion), y el nombre de la función que será invocada dentro del código objeto previamente definido (ejem. `suma`). El directorio donde se debe de copiar nuestro código objeto (.so) puede encontrarse con el comando `pg_config --pkglibdir`.

PostgreSQL al igual que MySQL no compilan funciones en C automáticamente, la UDF debe ser compilada fuera del SMBD y posteriormente se invoca la función mediante el comando CREATE FUNCTION.

Para asegurarse de que el código objeto no será implementado dentro de

una versión incompatible, PostgreSQL verifica que este contenga un "magic block", esto permite a PostgreSQL detectar incompatibilidades para versiones más recientes, por ejemplo, un "bloque mágico"es necesario para PostgreSQL 8.2 sino un error será devuelto. Para incluirlo se escribe el siguiente fragmento de código (teniendo la cabecera `fmgr.h`) :

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

la condición `#ifdef` prueba si el código no necesita ser compilado en versiones anteriores a PostgreSQL 8.2.

A continuación se muestra el Cuadro 4.2, el cual especifica la correspondencia de tipos (los más utilizados) en SQL y en C así como la cabecera que necesita ser definida para obtener la definición de estos tipos. Hay que notar que siempre tenemos que definir la cabecera `postgres.h` debido a que declara un número de funciones que son comúnmente utilizadas.

Tipo en SQL	Tipo en C	Definido en
boolean	bool	postgres.h
char	char	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2	postgres.h
integer (int4)	int4	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
point	POINT4*	utils/geo_decls.h
text	text*	postgres.h
time	TimeADT	utils/date.h
varchar	VarChar*	postgres.h

Cuadro 4.2: Correspondencia entre tipos de datos en SQL y C en PostgreSQL.

Ahora que tenemos una idea general de como funcionan las UDFs, pasamos a estudiar algunos ejemplos de funciones reales, asimismo, ir mostrando las características generales para su construcción.

Recordemos que utilizaremos las versión 1 para definir las UDFs. La declaración de la función en C es siempre como se muestra a continuación:

```
Datum nombre_de_la_funcion(PG_FUNCTION_ARGS)
```

Esta llama a la macro, la cual también se debe definir en el código:

```
PG_FUNCTION_INFO_V1(nombre_de_la_funcion);
```

Debe de aparecer en el mismo archivo fuente (.c), convencionalmente, debe ser escrita justo antes de *Datum nombre_de_la_funcion(PG_FUNCTION_ARGS)*. Luego para traer un argumento se utiliza la macro `PG_GETARG_xxx(n)` donde xxx corresponde al tipo de dato (int4, float4, float8, etc.) y *n* es la columna que se desea utilizar, siendo 0 la primera columna. Después el resultado de la función es devuelto a través de la macro `PG_RETURN_xxx()` donde xxx corresponde al tipo de dato a regresar. Pasamos a mostrar algunos ejemplos para aclarar lo anterior:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sumar_uno);

Datum sumar_uno(PG_FUNCTION_ARGS)
{
int32 arg = PG_GETARG_INT32(0);
PG_RETURN_INT32(arg+1);
}

PG_FUNCTION_INFO_V1(cuadrado);


Datum cuadrado( PG_FUNCTION_ARGS )
{
float4 *arg = (float4 *) palloc(sizeof(float4));
```

```
*arg = PG_GETARG_FLOAT4(0);
float4 resultado= pow((*arg),2);


PG_RETURN_FLOAT4(resultado);


}

PG_FUNCTION_INFO_V1(concatenar_texto);
Datum concatenar_texto(PG_FUNCTION_ARGS)
{
text *arg1 = PG_GETARG_TEXT_P(0);
text *arg2 = PG_GETARG_TEXT_P(1);
int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
text *new_text = (text *) palloc(new_text_size);
VARATT_SIZEP(new_text) = new_text_size;
memcpy(VARDATA(new_text), VARDATA(arg1),
VARSIZE(arg1) - VARHDRSZ);
memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
PG_RETURN_TEXT_P(new_text);
}
```

`VARHDRSZ` es lo mismo que `sizeof(int4)`, pero es considerado un buen estilo el utilizar esta macro para referirse al tamaño del overhead para un tipo de longitud variable. Por otro lado, cuando reservamos memoria se utiliza la función de PostgreSQL `palloc` en lugar de su correspondiente en C, `malloc` y `free`.

Vemos que la primera función devuelve el mismo número de filas pero suma uno a cada valor ingresado. La segunda función devuelve el cuadrado de cada valor ingresado, y por último, la tercera función concatena los valores ingresados fila por fila.

Supongamos que las funciones de arriba están en un archivo *Funciones.c* y es compilado a código objeto llamado Misfunciones.so, asimismo, este código objeto fue copiado al directorio donde PostgreSQL guarda todas sus bibliotecas compartidas, en nuestro caso fue: /usr/lib/postgresql/8.2/lib y una vez

hecho esto podemos registrar las funciones en PostgreSQL como se muestra a continuación:

```
CREATE FUNCTION add_one(integer) RETURNS integer
AS ´Misfunciones´, ´sumar_uno´
LANGUAGE C STRICT;


CREATE FUNCTION cuadrado(real) RETURNS real
AS ´Misfunciones´, ´cuadrado´
LANGUAGE C STRICT;


CREATE FUNCTION concatenar_texto(text, text) RETURNS text
AS ´Misfunciones´, ´concatenar_texto´
LANGUAGE C STRICT;
```

Notemos que hemos especificado las funciones como: "strict", esto significa que el sistema debería automáticamente asumir un resultado nulo si cualquier valor de entrada es nulo. Al hacer esto evitamos tener que analizar la existencia de valores nulos dentro del código de la función. Volviendo al tema de las ventajas de utilizar la versión 1, esta permite un mejor manejo para valores de entrada y resultados nulos. La macro `PG_ARGISNULL(n)` permite a una función probar si cada entrada es un valor nulo (desde luego que esto es sólo cuando no hayamos declarado la función como "strict"). Debemos abstenernos de ejecutar `PG_GETARG_xxx(n)` hasta que hayamos verificado que los valores de entrada no sean nulos. Para devolver un valor nulo, se ejecuta la macro `PG_RETURN_NULL()`; esto funciona tanto en funciones con la cláusula "strict" y sin ella.

A continuación listamos algunas reglas básicas que debemos tener en cuenta al momento de construir UDFs en PostgreSQL:

- Usar `pg_config --includedir-server` para localizar donde esta la carpeta que contiene las cabeceras necesarias para compilar nuestras funciones (ejem. postgres.h).
- Compilar y copiar el código objeto a la carpeta donde PostgreSQL contiene sus bibliotecas compartidas.

- Recordar definir un “magic block”para el código objeto (o biblioteca compartida).
- Cuando deseemos reservar memoria, usar la función proporcionada por PostgreSQL; `palloc`, en lugar de sus correspondientes en C; `malloc` y `free`. La memoria reservada por `palloc` será liberada automáticamente al final de cada transacción.
- Muchos de los tipos definidos en PostgreSQL están definidos en la cabecera `postgres.h`, mientras que las funciones que permiten administrar las interfaces (ejem. `PG_FUNCTION_ARGS`) están en `fmgr.h`, entonces debemos al menos incluir estas dos cabeceras, Por razones de portabilidad se recomienda incluir `postgres.h` al principio del código.
- El nombre del código objeto no debe de ser igual a ningún otro dentro de la carpeta de bibliotecas compartidas de PostgreSQL, así evitaremos posibles conflictos de renombramiento.

Por último presentamos la forma en que deben ser compiladas y enlazadas las UDFs para poder producir el código objeto que será cargado dinámicamente por PostgreSQL. Para evitar confusiones a partir de ahora utilizaremos nuevamente el nombre de biblioteca compartida en lugar de código objeto, esto para no confundirnos con el archivo objeto que veremos a continuación. Entonces el objetivo es producir una biblioteca compartida, primero el archivo fuente (.c) es compilado a un archivo objeto, esto se hace de la siguiente manera:

```
gcc -c Funciones.c -I /usr/include/postgresql/8.2/server/
```

Recordemos que el directorio puede ser localizado por los comandos `pg_config --includedir-server`, para nuestro caso este directorio fue el que utilizamos para compilar el código fuente. Después los archivos objeto son enlazados entre ellos, esto se logra con los comandos siguientes:

```
gcc -fpic -c Funciones.c -I /usr/include/postgresql/8.2/server/
```

Por último creamos las biblioteca compartida con los siguientes comandos:

```
gcc -shared -o Misfunciones.so Funciones.o
-I /usr/include/postgresql/8.2/server/
```

Una vez que tenemos nuestra biblioteca compartida lo último que nos resta por hacer es copiarla al directorio donde PostgreSQL tiene sus demás bibliotecas compartidas, recordemos que este directorio lo podemos encontrar con el comando `pg_config --pkglibdir`. Una vez que sabemos donde copiarlo sólo ponemos el siguiente comando como root (por ser Linux):

```
cp Misfunciones.so /usr/lib/postgresql/8.2/lib
```

La biblioteca compartida resultante ahora puede ser cargada dentro de PostgreSQL a través del comando CREATE FUNCTION, como se vio anteriormente. Por último, si deseamos utilizar nuestra función recién creada sólo tenemos que escribir una simple sentencia en SQL por ejemplo:

```
SELECT cuadrado(atributox) from tabla;
```

Donde la función `cuadrado` fue previamente definida.

Como vimos la forma de compilar es similar a MySQL, sin embargo, la forma en que son construidas las UDF así como las macros y funciones que cada uno implementa hace difícil la portabilidad entre ambos manejadores.

4.2.2. UDFs agregadas

Las funciones de agregación en PostgreSQL son expresadas en términos de *valores de estado* (*state values*) y de *funciones de transición* (*transition functions*). Esto es, una función de agregación opera utilizando un valor de estado el cual es actualizado por cada fila de entrada. Para definir una nueva función de agregación, debemos de seleccionar un tipo de dato para el valor de estado, un valor inicial para el estado, y una función de transición. La función de transición es una UDF que sigue las mismas reglas para su construcción mencionadas anteriormente. Una *función final* puede ser especificada, para el caso donde el resultado deseado sea diferente al valor devuelto por la función de transición.

Una función de agregación es identificada por su nombre y el tipo de entrada. Puede haber dos o más funciones de agregación con el mismo nombre si

operan sobre diferentes tipos de entrada.

Como mencionamos anteriormente, una función de agregación esta construida de una o dos UDFs: una función de transición *sfunc*, y una función final opcional *ffunc*. Estas son utilizadas como sigue:

sfunc(estado-interno, campo)
ffunc(estado-interno)

El *estado interno* representa la salida que utiliza internamente la función de transición así como la salida de la misma. Entonces, primero se agregan los valores de cada fila ingresada hasta que se procesan todas las tuplas de la tabla o base de datos, una vez que se tienen los valores agregados (los cuales representan a los estados internos) se invoca a la función de final (si esta es requerida) **una sola vez** y se pasan como entrada los estados internos devueltos por la función de transición. Si no hay función final entonces el valor final es devuelto como es, como ejemplo sería la suma, la cual no necesita de un calculo adicional para realizar esta operación. Para hacer lo anterior, PostgreSQL crea una variable temporal *stype* para mantener el valor actual de la agregación. Además, para que PostgreSQL pueda saber si es una función de transición o una función final debemos de poner la siguiente linea de código[7] en nuestra UDF que corresponde a la función de transición:

```
if (fcinfo->context && IsA(fcinfo->context, AggState))
```

La razón de verificar esto es que cuando es verdadera, la primera entrada debe ser un valor temporal y por lo tanto, puede ser modificado de una manera segura.

Si la función de transición es declarada como “strict”, entonces no será invocada con entradas nulas. Esta función de transición se comporta de la manera siguiente: Los valores nulos son ignorados (la función no es invocada y el valor previo es retenido). Si el valor inicial es nulo, entonces el primer valor no nulo reemplazará este valor, y la función de transición es invocada empezando con el segundo valor no nulo como entrada.

[7]Para ver con más claridad esto véase los ejemplos en el apéndice B.

Si la función de transición es no estricta, entonces se invocará incondicionalmente por cada valor de entrada, y debemos de tratar a los valores nulos dentro de las UDFs, esto nos permite tratar a los nulos como queramos. Sin embargo, en este trabajo definiremos a las funciones como estrictas.

Para aclarar todo lo anterior veamos algunos ejemplos. Si deseamos definir una función agregada que no utiliza una función final, esto es, sólo definiendo una función de transición, la cual fue construida como se mostró en la Sección 4.2 (para ver el código de esta función véase el apéndice B). Antes de definir la función de agregación, definiremos cómo debemos registrar las UDFs en PostgreSQL para entender de una manera más clara:

```
CREATE FUNCTION suma(float8, real) RETURNS float8
AS ´Misfunciones´, ´suma´
LANGUAGE C STRICT;
```

Como podemos observar, la función *suma* es definida con dos parámetros de entrada, esto es debido a que el primer parámetro (float8) es el estado interno el cual va guardando el valor agregado, para posteriormente convertirse en la salida que devolverá la función. Por otro lado, el segundo parámetro representa el tipo de entrada que aceptará la función y el cual deberá tener el campo en donde deseemos aplicar esta función. Esta es la principal diferencia al definir una UDF que servirá como una función de transición (al definir dos parámetros) para construir una función de agregación. Una vez que definimos la función de transición (suma) pasamos a definir la función de agregación la cual se ve de la siguiente manera:

```
CREATE AGGREGATE misuma (real)
{
SFUNC = suma,
STYPE = float8
INITCOND= '0'
};
```

Donde *sfunc* es el nombre de la función de transición, *stype* es el tipo de salida que devolverá la función de transición, e *initcond* es la función inicial

que tendrá la función de transición, asimismo, nos permite definir el formato de salida (que sea valido para el tipo de dato). Para este ejemplo nos permitirá devolver un solo valor.

Entonces una vez definida la función agregada podemos invocarla a través de cualquier sentencia "SELECT" por ejemplo:

```
SELECT misuma (a) FROM tablaX;
```

Ahora bien, para el caso donde se necesite una función final para realizar un calculo posterior se tienen que definir dos UDFs: Una correspondiente a la función de transición y otra a la función final. A continuación veremos un ejemplo más complejo: la correlación de dos atributos. Para esto, volvemos a partir de la manera de registrar las UDFs en PostgreSQL y posteriormente definimos la función de agregación para la correlación:

```
CREATE FUNCTION acumulacorr(float8[ ], real, real) RETURNS float8[ ]
AS ´Misfunciones´, ´acumulacorr´
LANGUAGE C STRICT;
```

```
CREATE FUNCTION fcorr(float8[ ]) RETURNS float8
AS ´Misfunciones´, ´fcorr´
LANGUAGE C STRICT;
```

Como podemos observar la primera función corresponde a la función de transición, vemos que ésta tiene tres parámetros de entrada, el primero es para guardar los valores agregados de la salida, el cual será un arreglo de seis elementos (esto debido a que se necesitaran para el calculo final), estos seis elementos se convertirán en la entrada de la segunda función, correspondiente a la función final, la cual procesará estos valores agregados y devolverá el valor final (el valor de la correlación), como podemos observar en la segunda función (final) sólo tiene un parámetro de entrada, asimismo, podemos observar que devolverá un valor de tipo float8, el cual corresponderá al valor de la correlación. Una vez que estudiamos y definimos las UDFs respectivas, pasamos a registrar la función de agregación para la correlación:

```
CREATE AGGREGATE micorrelacion (real,real)
{
SFUNC = acumulacorr,
STYPE = float8[ ]
FINALFUNC=fcorr
INITCOND= '{0,0,0,0,0,0}'
};
```

Como podemos observar, la función de agregación para la correlación necesitará dos parámetros de entrada de tipo real, para la función de transición correspondiente a la UDF *acumulacorr*, la salida es un arreglo de seis elementos (STYPE), la función final es la UDF *fcorr* la cual recibe este arreglo, hace los cálculos respectivos con estos valores agregados y posteriormente devuelve el valor final, por último, la condición inicial es un arreglo de seis elementos, los cuales corresponden a los seis elementos que devuelve la función de transición permitiendo así, definir una arreglo de salida con valor inicial 0 en cada elemento (para cuando no tengan valor). Esta función de agregación es invocada a través de cualquier sentencia "SELECT" como en el ejemplo anterior.

Como podemos observar con estos ejemplos, la manera de definir las UDFs es la misma que como vimos anteriormente donde definimos UDFs para operaciones escalares, con la diferencia de que para las UDFs correspondientes a las funciones de transición se define un parámetro extra que guardará los valores agregados. Estas UDFs se compilan exactamente igual como se explico en la Sección 4.2. Entonces, por último, hacemos un resumen de los pasos a seguir para construir una UDF agregada (o UDA) en PostreSQL.

- Primero debemos crear las UDFs correspondientes a la función de transición y la función final. Esto lo logramos estudiando la Sección 4.2.
- Posteriormente registramos las funciones antes mencionadas dentro de PostgreSQL. Esto a través de la sentencia CREATE FUNCTION.
- Definimos la función de agregación, la cual utilizará la función de transición y la función final (si es requerida) previamente definidas.
- Invocamos a esta función de agregación a través una sentencia "SELECT" para su uso normal.

En el apéndice B ponemos el código completo para construir dos funciones de agregación, estos ejemplos se pueden tomar como base para construir muchas más funciones de agregación.

Para observar la referencia completa de la forma de construir UDFs escalares y agregadas en PostgreSQL véase [16].

4.3. SQL Server 2005

4.3.1. UDFs escalares

En esta sección explicaremos como las UDFs son implementadas en SQL Server 2005. Iniciamos como lo hemos venido haciendo: explicando las UDFs que tratan funciones escalares, esto es, funciones que devuelven un valor por cada tupla en la tabla o base de datos y posteriormente estudiaremos las UDFs agregadas.

Una característica de SQL Server 2005 es su integración del *entorno de ejecución común de .NET*[8] (CLR, Common Language Runtime), esta integración nos permite crear UDFs utilizando lenguajes orientados a objetos como: Visual Basic .NET y *C#*, aunque también podemos utilizar $C++$. En lo que respecta a esta sección los ejemplos que mostremos serán hechos en *C#*[9], aunque también se pueden construir en $C++$ y Visual Basic .NET sin ningún problema.

En versiones anteriores de SQL Server 2005, se estaba limitado a utilizar sólo SQL. Con la integración de CLR, podemos ahora realizar las tareas que eran imposibles o difíciles de alcanzar con sólo SQL. Tanto Visual Basic .NET como *C#* ofrecen el apoyo total para arreglos, el manejo de las excepciones, y las colecciones. Con estos lenguajes, podemos aprovechar la integración de CLR para escribir código más complejo y que realice distintas tareas satisfactoriamente. Visual Basic .NET y *C#* ofrecen capacidades orientadas

[8]Es el motor en tiempo de ejecución de .NET que ejecuta todos los programas.

[9]Esto a efecto de mostrar el comportamiento del desempeño de las funciones de agregación con otro lenguaje de programación.

a objetos tales como: encapsulación, herencia, y polimorfismo. El código se puede organizar en clases. Además gracias a estos lenguajes, y el framework base de .NET, tenemos acceso a miles de clases pre-construidas así como de rutinas [8].

Tipos de datos

Los tipos de datos que se pueden utilizar son: para las entradas son todos los tipos excepto *text, ntext, image* y *timestamp*, por otro lado, para la salida se permiten todos los tipos de datos, excepto *timestamp*. La Tabla 4.3 muestra a detalle la equivalencia de tipos:

SqlTypes de .NET Framework	**Tipo en SQL**
SqlBinary	binary, varbinary
SqlBoolean	bit
SqlByte	tinyint
SqlDateTime	datetime, smalldatetime
SqlDecimal	decimal
SqlDouble	float
SqlGuid	uniqueidentifier
Sqlint16	smallint
Sqlint32	int
Sqlint64	bigint
SqlMoney	money, smallmoney
SqlSingle	real
SqlString	char, nchar, nvarchar, varchar
SqlXml	xml

Cuadro 4.3: Correspondencia entre tipos de datos en SQL y .NET (C++, *C#* y VB .NET) en SQL Server 2005.

¿Qué es un *Assembly* (ensamblado)?

Un *assembly* es un modulo de aplicación administrada que contiene metadatos de clase y código administrado como un objeto en una instancia de SQL

Server. Mediante este módulo, podemos crear dentro de la base de datos funciones CLR. Para crear los objetos, debemos escribir el código correspondiente que implemente la funcionalidad de la UDF. Una vez que el código es escrito, debemos compilarlo dentro de un *assembly* de .NET y luego enlazarlo con SQL Server 2005. Esto se puede realizar de dos formas distintas:

1. La forma más simple es utilizar Visual Studio 2005, en donde creamos un nuevo proyecto para SQL Server. Después de crear el proyecto, podemos crear la UDF y enlazarla dentro de SQL Server para posteriormente poder utilizarla a través de cualquier sentencia "SELECT". De este modo eliminamos los pasos de registro manual de la UDF en SQL Server.

2. Creamos un proyecto de biblioteca de clase de Visual Studio y lo compilamos dentro de un *assembly*. Una vez que el *assembly* es creado, debemos registrarlo en SQL Server y entonces asociar la definición de la UDF con el método (o función) contenido en el *assembly*.

Como podemos observar la segunda opción es la alternativa manual de como crear, compilar y enlazar las UDFs con SQL Server. En cambio, la primera alternativa es un proceso automático que elimina muchos de estos pasos, a continuación explicamos más a detalle estas alternativas.

Implementando UDFs utilizando Visual Studio 2005

Para empezar, creamos un nuevo proyecto de SQL Server utilizando el menú de Visual Studio 2005, por ejemplo, `File` → `New` → `Project` → `Visual` *C#* → `Database` → `SQL Server Project`. Una vez aquí especificamos el nombre del proyecto, por ejemplo *MIUDF*, en este proyecto crearemos una UDF que calcule el producto de dos atributos, tupla por tupla.

Debido a que creamos un proyecto de base de datos, necesitamos asociar una base de datos (creada en SQL Server previamente) con nuestro proyecto. Para este fin, Visual Studio nos propone seleccionar una base de datos existente o agregar una nueva base de datos[10]. Por ejemplo, escogemos una base de

[10]Sólo es la referencia de la base de datos, no nos permite crear una base de datos, esto se tiene que hacer en SQL Server 2005.

datos llamada TPCH.

Una vez que hemos creado el proyecto, nos dirigimos a `Project` → `Add` → `User-Defined Function` del menú de Visual Studio. En el cuadro de dialogo *Add New Item*, introducimos el nombre de nuestra UDF, en nuestro ejemplo será *prod.cs* y presionamos el botón *add*.

Después de haber creado la clase, aparecerá un código similar a este:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString prod()
    {
        // Put your code here
        return new SqlString("Hello");
    }
};
```

Como podemos observar, Visual Studio nos da la estructura general de como se deben de construir las UDFs así como proporciona las bibliotecas necesarias y más utilizadas. Entonces el código para realizar el producto quedaría de la siguiente manera:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
```

```
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlDouble prod(SqlDouble x, SqlDouble y)
   {
      return x * y;
   }
};
```

El código anterior comienza importando los *namespaces* (bibliotecas) requeridos. Después de esto, el código declara un método estático nombrado *prod* que admite dos parámetros. El método *prod* es adornado con el atributo [Microsoft.SqlServer.Server.SqlFunction], que especifica que el método *prod* será expuesto como UDF. En el método *prod*, simplemente se devuelve el resultado del producto de las dos entradas. Cabe señalar que una UDF que devuelve tipos escalares debe devolver tipos de datos .NET que puedan ser implícitamente convertidos a un tipo de SQL Server. En este ejemplo las entradas serán de tipo float y la función devolverá un valor de tipo float.

Ahora que hemos construido la UDF, el enlazamiento con SQL Server es muy simple y directo. Antes de enlazarla UDF, necesitamos compilar el proyecto, esto se logra al presionar la opción `Built` → `Built MIUDF` del menú de Visual Studio. Esto compilará todas las clases del proyecto, y en caso de ocurrir un error de compilación éste será desplegado en la lista de errores.

Una vez que el proyecto ha sido compilado, podemos enlazarlo con SQL Server al seleccionar la opción `Build` → `Deploy MIUDF` del menú. Esto no sólo registrará el *assembly* sino además enlazará la UDF con SQL Server.

Una característica interesante que tenemos al momento de crear UDFs en SQL Server mediante Visual Studio 2005, es que nos permite probar nuestras funciones construidas. Entonces, para realizar esto debemos seleccionar la opción `View` → `Server Explorer` del menú de Visual Studio, en el nodo de *Data Connections*, abrimos la conexión a la base de datos que especificamos anteriormente, luego, abrimos el nodo *Functions*. Apretamos el botón derecho sobre este nodo y seleccionamos *Execute* del menú contextual[11]. Aquí aparecerá un cuadro de dialogo que permitirá introducir datos de

[11]Hasta este punto las UDFs agregadas pueden ser igualmente (no construidas) invo-

ejemplo en el campo *Value*, una vez que introduzcamos los valores apretamos `ok`, y aparecerá el resultado en la salida (output) de Visual Studio.

Una vez que probamos que nuestra UDF funciona correctamente habremos terminado el proceso de construcción de UDFs en SQL Server. Solo resta mostrar la manera de invocarlas en SQL Server 2005. Esto se hace de la siguiente manera:

```
SELECT dbo.prod(X, Y) FROM tabla;
```

Como podemos observar con este ejemplo, el uso de Visual Studio brinda una gran ayuda al momento de construir las UDFs, así como compilarlas y enlazarlas con SQL Server 2005. Este procedimiento también puede ser realizado para crear UDFs en Visual Basic .NET y en $C++$, claro con unas pequeñas modificaciones en el código.

Segunda alternativa de implementar UDFs

La manera manual de registrar y enlazar las UDFs consta de los siguientes pasos:

1. Crear la clase que implementa la funcionalidad de nuestra UDF.
2. Compilar la clase utilizando el compilador de .NET para producir un *assembly*.
3. Registrar el *assembly* con SQL Server utilizando la sentencia CREATE ASSEMBLY.
4. Asociar la definición de la UDF con el método especificado en la clase.

Los dos primeros puntos se realizan de la misma forma como acabamos de explicar. Ahora explicamos como se define la UDF manualmente en SQL

cadas, compiladas y enlazadas. Pero no podemos probar las UDFs agregadas en Visual Studio, esto debe ser directamente en SQL Server.

Server[12]. Entonces una vez que hemos construido y compilado nuestro proyecto, la clase que realiza la funcionalidad deseada genera un *assembly* que contiene el método que deseamos implementar, luego nos dirigimos al ambiente de trabajo de SQL Server 2005 en donde podemos cargar este *assembly* mediante el comando CREATE ASSEMBLY. Por ejemplo:

```
CREATE ASSEMBLY Mi_ensamblado FROM
´C:\Visual Studio 2005\Projects\MIUDF\MIUDF\bin\Debug\MIUDF.dll´
```

El comando CREATE ASSEMBLY toma un parámetro que contiene la ruta del *assembly* que definiremos dentro de SQL Server. Esto puede ser una ruta local (como en el ejemplo) o una ruta a un archivo compartido en la red. Cuando el comando CREATE ASSEMBLY es ejecutado, el *assembly* es copiado dentro de la base de datos maestra [10].

Después de registrar el *assembly*, podemos borrarlo en caso de ser necesario utilizando el comando DROP ASSEMBLY:

```
DROP ASSEMBLY Mi_ensamblado;
```

Debido a que un *assembly* permanece en la base de datos cuando el código fuente para ese *assembly* cambia, debemos primero borrar el *assembly* existente y posteriormente volver a registrar el mismo *assembly* para que se actualicen los cambios.

Hasta el momento hemos completado los pasos de: creación, compilación y el registro del *assembly*. Ahora, necesitamos asociar la función que será creada en SQL Server con el método apropiado dentro del *assembly* previamente registrado. A continuación se muestra una forma general de definir las funciones en SQL Server:

```
CREATE FUNCTION Nombre:funcion(lista de parámetros)
RETURNS tipo_de_salida
WITH opciones_para_nulos
```

[12] El equivalente a esto en la primera alternativa es cuando presionamos la opción *deploy*.

```
AS
EXTERNAL NAME
nombre_assembly.nombre_clase.nombre_metodo
```

El siguiente ejemplo ilustra como podemos utilizar el comando CREATE FUNCTION para crear nuestra UDF:

```
CREATE FUNCTION prod(@x float,@y float)
RETURNS float
AS
EXTERNAL NAME
Mi_ensamblado.Prod.Prod
```

Al ejecutar esta función obtendremos el mismo resultado que la primera alternativa (implementando el *deploy*), en donde la función recibirá dos parámetros de entrada de tipo float, asimismo, devolverá un valor escalar de tipo float. La cláusula EXTERNAL NAME, enlaza el nombre de la UDF con el método apropiado del *assembly*. En este ejemplo, el método *prod* esta en la clase *prod*, la cual a su vez esta en el *assembly* llamado Mi_ensamblado.

Por último, lo que nos resta explicar es la cláusula WITH, la cual nos permite especificar que deseamos hacer ante la presencia de valores nulos. Hay dos alternativas para esta cláusula:

- CALLED ON NULL INPUT (por defecto). Especifica que aún cuando la función sea invocada con valores nulos como argumentos, el cuerpo de la función será ejecutada de manera normal.
- RETURNS NULL ON NULL INPUT. Si se especifica esta opción, indica que SQL Server puede devolver NULL cuando cualquiera de los argumentos que reciba sea NULL, sin invocar realmente el cuerpo de la función.

Como podemos observar al realizar el registro de nuestra UDF manualmente tenemos acceso a más opciones que podríamos utilizar para tener un mejor

control sobre nuestros resultados, cosa que es difícil con la primera alternativa, entonces el uso de cualquiera de estas dos alternativas dependerá del conocimiento y los requerimientos necesarios.

Con esto terminamos nuestro estudio de las UDFs escalares en SQL Server 2005, cabe mencionar que el SMBD que tiene más información sobre las UDFs es éste, asimismo, también para las UDFs agregadas que veremos a continuación. Entonces si se desean ver más ejemplos de UDFs implementadas en SQL Server 2005 véase [32, 10, 8].

4.3.2. UDFs agregadas

Las funciones de agregación intrínsecas en SQL Server 2005 son: COUNT(), SUM(), AVG(), MAX(), MIN() y STANDARD DEVIATION(). Sin embargo, frecuentemente este conjunto de funciones no son suficientes. Entonces cuando deseemos implementar agregaciones más complejas, podemos lograr esto construyendo consultas complejas en SQL, sin embargo, la dificultad de crear y mantener estas consultas es muy costoso. Las UDFs agregadas (o UDAs) nos permiten desarrollar expresiones de esta complejidad de una manera simple, al implementar el poder de la biblioteca de clase base de .NET (BLC) y el poder de los lenguajes tales como: $C\#$, $C++$ y Visual Basic .NET. Para posteriormente ser implementadas como las funciones intrínsecas y utilizadas de la misma manera [10].

A continuación se muestran algunas limitaciones que tienen las UDFs agregadas en SQL Server 2005:

- El tamaño de las UDFs agregadas tiene un limite de 8000 bytes. Si una instancia de nuestra UDF agregada excede los 8000 bytes, una excepción será devuelta deteniendo el procesamiento de la consulta.
- Las UDFs agregadas no soportan parámetros directamente, esto es, sólo aceptan la columna de datos que le pasamos al acumulado. Para lograr construir una UDF agregada (con más de un parámetro de entrada) se tiene que realizar de una manera indirecta (como se verá al final de esta sección).

Construyendo una UDF agregada

Para implementar una UDF agregada debemos utilizar una clase o estructura, por defecto, cuando añadimos una nueva agregación en nuestro proyecto en Visual Studio, se crea una plantilla la cual contiene un conjunto de métodos que forman el ciclo de vida de una UDF agregada. A continuación mostramos los métodos que forman el contrato de la agregación:

- **INIT()**: Este método es invocado cuando se inicia la agregación. La inicialización de variables de la agregación se definen aquí.
- **ACCUMULATE()**: Como cada fila es procesada, esto incluye la primera y ultima fila, este método es invocado para sumar estos valores en la agregación. Por ejemplo, si estuviéramos escribiendo una agregación que contara el número de filas, este método simplemente incrementaría un contador interno. Pero si la agregación fuera el promedio, éste incrementaría el contador y la suma de los valores de las filas.
- **MERGE()**: El optimizador de SQL Server 2005 puede decidir si multiples procesadores son presentados en el servidor. Si esto ocurre para una operación de agregación, este método puede ser automáticamente utilizado. Dos o más agregaciones parciales pueden ser creadas en paralelo, antes de invocar el método TERMINATE(), cualquier agregación parcial será fusionada utilizando este método.
- **TERMINATE()**: Una vez que todas las filas han sido procesadas, este método es invocado. Este método devuelve el resultado final de la UDF agregada.

Construyendo una UDF agregada

Las UDFs agregadas deben ser definidas dentro de un *assembly* (al igual que las UDFs escalares). A continuación mostramos un ejemplo el cual cuenta las filas (como la función COUNT()) para aclarar lo anterior. La clase para esta UDF agregada requiere que cada uno de los cuatro métodos este presente, aunque sólo una variable privada será requerida, la cual será utilizada para mantener la cuenta.

Para empezar, crearemos un nuevo proyecto de base de datos dentro de Visual Studio 2005[13] (como se hizo en la Sección 4.3), pero ahora lo llamaremos *MIUDA*, y añadimos una agregación (`Project` → `Add Aggregate`), la cual llamaremos *cuenta*. La siguiente plantilla aparecerá al haber creado la clase anterior:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.
                   SqlUserDefinedAggregate(Format.Native)]
public struct cuenta
{
    public void Init()
    {
        // Put your code here
    }
    public void Accumulate(SqlString Value)
    {
        // Put your code here
    }
    public void Merge(cuenta Group)
    {
        // Put your code here
    }
    public SqlString Terminate()
    {
        // Put your code here
        return new SqlString("");
    }
    // This is a place-holder member field
```

[13]Cabe aclarar que aunque en esta sección sólo explicaremos las UDF agregadas en $C\#$, estas también pueden ser construidas tanto en $C++$ como en Visual Basic .NET.

```
    private int var1;
}
```

Como podemos observar esta plantilla al igual que las UDFs escalares, tienen los *namespaces* (bibliotecas) necesarios para implementar nuestras agregaciones. Asimismo, una UDF agregada debe ser decorada con un atributo llamado *SqlUserDefinedAggregate*. Este atributo es utilizado por SQL Server para mostrar que el *assembly* es una agregación y verifique que el contrato contenga los cuatro métodos mencionados anteriormente. Este atributo tiene algunas opciones que están desactivadas por defecto (FALSE), las cuales mencionamos a continuación:

- **IsInvariantToDuplicates**: Este parámetro determina si nuestra UDF agregada devolverá el mismo resultado sin importar si valores duplicados son ingresados.

- **IsInvariatToNulls**: Este parámetro determina si nuestra UDF agregada devolverá el mismo resultado a pesar de la existencia de valores nulos.

- **IsNullIfEmpty**: Este parámetro determina si nuestra UDF agregada devolverá NULL cuando ninguna fila ha sido ingresada.

Estas opciones nos permiten tener un mejor control de nuestras UDFs agregadas (ejem. SqlUserDefinedAggregate(Format::Native, IsInvariantToDuplicates:=True).

La opción *serializable* indica que los datos de la clase serán almacenados en formato serial. Posteriormente dentro de esta plantilla aparecen los cuatro métodos antes mencionados (INIT(), ACCUMULATE(), etc) y por último aparece el miembro *private* en donde definiremos las variables que utilizaremos, no hay que confundirnos con el método INIT() en donde se inicializan estas variables. Ahora que explicamos la plantilla que aparece por defecto, pasamos a mostrar el código que implementa la cuenta:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.
                    SqlUserDefinedAggregate(Format.Native)]
public struct cuenta
{
    public void Init()
    {
        Micuenta = 0;
    }
    public void Accumulate(SqlDouble Value)
    {
        Micuenta += 1;
    }
    public void Merge(cuenta Group)
    {
        Micuenta += Group.Micuenta;
    }
    public SqlInt64 Terminate()
    {
        return Micuenta.ToSqlInt64();
    }
    // This is a place-holder member field
    private SqlDouble Micuenta;
}
```

Como podemos observar, el método INIT() contiene la variable Micuenta, la cual inicializamos a 0. Posteriormente el método ACCUMULATE() el cual recibe un parámetro de tipo double (correspondiente a la columna que se va contar), aquí la variable *Micuenta* va ir actualizando la cuenta de las

filas conforme se vayan procesando, el método MERGE() acumula todos los grupos (sólo en caso de instrumentar esta agregación en paralelo), este paso garantizará que tengamos todos los valores contados aún cuando hayamos ejecutado la agregación en paralelo. El método TERMINATE() es utilizado para devolver el valor acumulado final, correspondiente a la cuenta final (el cual será un entero).

Registrando una UDF agregada

Ahora que ya sabemos como se construyen las UDFs agregadas pasamos a registrar el *assembly* que contiene a nuestra UDF agregada. Al igual que las UDFs escalares, existen dos maneras de hacer esto: La forma directa, en donde con ayuda de la opción *deploy* podemos definir la agregación directamente en SQL Server y la forma manual. Para dejar esto más claro explicaremos la forma manual de como definir las UDFs agregadas en SQL Server 2005, dejando la forma directa al lector (véase UDFs escalares). Entonces una vez que hemos compilado nuestro proyecto junto con sus clases (con la opción *built*) nos pasamos a SQL Server 2005 (SQL Server Management Studio) en donde ejecutaremos las siguientes sentencias:

```
USE TPCH
 GO

 CREATE ASSEMBLY Micuenta
 FROM ´C:\Visual Studio 2005\Projects\MIUDA\debug\MIUDA.dll´
 GO

 CREATE AGGREGATE Cuenta
 ( @Value float )
 RETURNS Bigint
 EXTERNAL NAME Micuenta.cuenta
```

Supongamos que nuestra base de datos se llama TPCH, entonces definimos el *assembly* con el nombre *Micuenta*, este *assembly* es enlazado del directorio arriba mencionado en donde se encuentra el archivo .dll, en este archivo

se encuentra el método que realiza nuestra agregación. Entonces, una vez registrado el *assembly* pasamos a definir nuestra agregación mediante el comando CREATE AGGREGATE, aquí definimos el tipo de entrada permitida (float) así como el tipo de salida (bigint), posteriormente en la cláusula EXTERNAL NAME especificamos el método que realiza nuestra agregación (el cual esta precedido por el nombre del *assembly* que lo contiene), el cual para nuestro ejemplo es *cuenta*. Entonces, una vez que hemos registrado nuestra agregación en SQL Server 2005, lo que nos resta por hacer es invocarla a través de una sentencia SELECT, como lo hemos hecho hasta ahora:

SELECT dbo.Cuenta(columnaX) FROM tablax;

Como podemos observar, construir UDFs agregadas en SQL Server 2005 es más directo y más sencillo (gracias a Visual Studio 2005) que PostgreSQL y MySQL, pero hasta ahora sólo hemos tratado agregaciones que involucran sólo un parámetro de entrada, para lograr UDFs agregadas que soporten más de un parámetro de entrada tenemos que seguir una serie de pasos adicionales. Explicaremos este proceso con un ejemplo, la función de correlación, la cual requiere dos parámetros de entrada, pero antes mencionaremos los pasos generales a seguir para realizar esto:

1. Creamos un Tipo Definido por el Usuario (UDT por sus siglas en ingles), el cual permite ingresar y devolver un valor de tipo con dos parámetros.

2. Creamos una UDF escalar, en la cual ingresaremos los dos parámetros de entrada (esto debido a que las UDF escalares si permiten multiples parámetros de entrada).

3. Por último creamos la UDF agregada que tenga como parámetro a la UDF creada en el paso 2 y como tipo de datos el creado en el paso 1.

Explicaremos los puntos anteriores en el mismo orden.

Tipos Definidos por el Usuario (UDTs)

La idea de crear un nuevo tipo de datos es poder cubrir nuestras necesidades de una manera más satisfactoria. SQL Server 2005 con la tecnología CLR, nos permite crear UDTs de una manera sencilla y eficiente. Estas UDTs no necesariamente deben de estar basadas sobre los tipos nativos de SQL Server. A continuación mostramos el código para realizar el punto 1, el cual lo analizaremos de manera general[14]:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType
                     (Format.UserDefined, MaxByteSize=8000)]
public struct doblentrada : INullable, IBinarySerialize
{
    public override string ToString()
    {
            throw new Exception("The method or operation
                                  is not implemented.");
    }

    public bool IsNull
    {
        get
        {
            return false;
        }
    }

    public static doblentrada Null
```

[14]Si se desea saber más sobre UDTs véase [10].

```
    {
        get
        {
            throw new Exception("The method or operation
                                    is not implemented.");
        }
    }

    public static doblentrada Parse(SqlString s)
    {
        throw new Exception("The method or operation
                                    is not implemented.");
    }
    public string primera;

    public string segunda;

#region IBinarySerialize Members

    void IBinarySerialize.Read(System.IO.BinaryReader r)
    {
        primera = r.ReadString();
        segunda = r.ReadString();
    }
    void IBinarySerialize.Write(System.IO.BinaryWriter w)

    {
        w.Write(primera);
        w.Write(segunda);
    }
    #endregion
}
```

La creación de un UDT en Visual Studio es similar a lo que hemos venido haciendo hasta ahora (`Project` → `Add User Defined Type`), en donde aparecerá una plantilla la cual fue modificada hasta quedar como se muestra arriba. El atributo **SqlUserDefinedType** nos permite definir algunas propiedades

que gobernarán al UDT, por ejemplo, **UserDefined** es utilizado cuando nuestro UDT requiere una referencia de tipo cadena. Cuando implementamos este atributo también debemos definir la interface **IBinarySerialize**, la cual añadirá dos funciones: *Read* y *Write*, estas funciones escribirán y leerán los valores de nuestro UDT. La interface **INullable** debe ser definida, esta interface sólo tiene una propiedad: *IsNull*. Está es definida como una salida booleana, la cual nos indica si o no el valor del UDT es un valor nulo.

Ahora pasamos a explicar los tres métodos que deben tener los UDTs:

- **ToString**. Este método es utilizado para convertir los valores de tipo a una representación de tipo cadena.
- **Parse**. Este es el segundo requerimiento para la conversión a tipo cadena, permite a una cadena ser convertida en un UDT. Es un método estático, el cual es invocado cuando los valores son asignados a una instancia del tipo.
- **NULL**. Este método es necesario para que el UDT pueda reconocer un valor nulo.

Hasta el momento hemos construido el código de la UDT, lo siguiente es compilar el proyecto con la opción *built*, y pasamos a registrar el *assembly* correspondiente en SQL Server 2005, esto se hace de la siguiente manera:

```
CREATE ASSEMBLY Miudt
FROM ´C:\Visual Studio 2005\Projects\MIUDT\debug\MIUDT.dll´
```

Lo siguiente es registrar el tipo a utilizar dentro de SQL Server, esto se realiza a través de la sentencia CREATE TYPE:

```
CREATE TYPE doblentrada
EXTERNAL NAME Miudt.doblentrada
```

Ahora que hemos construido nuestro UDT que contiene los campos requeridos para nuestra agregación, pasamos a construir la UDF correspondiente que

tomará un número de parámetros (dos para este ejemplo) y devolverá una instancia del UDT. A continuación se muestra el código de la UDF que realiza lo anterior:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
  [Microsoft.SqlServer.Server.SqlFunction]
  public static doblentrada Entbinaria(string primera,
                                            string segunda)
    {
        doblentrada o = new doblentrada();
        o.primera = primera;
        o.segunda = segunda;
        return o;
    }
};
```

Esta UDF toma dos entradas de tipo cadena y devuelve un UDT con dos parámetros, la cual se registra de la siguiente manera:

```
CREATE FUNCTION Entbinaria(@x String,@y String)
RETURNS doblentrada
AS
EXTERNAL NAME
MiUDF.Entbinaria
```

Supongamos que hemos definido el *assembly* llamado *MiUDF*, el cual contiene el método *Entbinaria*. Como podemos observar la salida es tipo *doblentrada*, el cual definimos previamente. Por último, creamos la UDF agregada (la

cual corresponde a la función de correlación) que tomará como parámetro de entrada la UDF recién creada. A continuación se muestra el código que implementa esto:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server
                    .SqlUserDefinedAggregate(Format.Native)]
public struct corr
{

    public void Init()
    {
        x = 0.0;
        y = 0.0;
        z = 0;
        sumxy = 0.0;
        sumxx = 0.0;
        sumyy = 0.0;
    }

    public void Accumulate(doblentrada o)
    {
        Double var1 = Double.Parse(o.primera);
        Double var2 = Double.Parse(o.segunda);

        x += var1;
        y += var2;
        z += 1;
        sumxx += Math.Pow(var1,2);
        sumxy += (var1) * (var2);
        sumyy += Math.Pow(var2, 2);
```

```
    }

    public void Merge(corr Group)
    {
        x += Group.x;
        y += Group.y;
        z += Group.z;
        sumxx += Group.sumxx;
        sumxy += Group.sumxy;
        sumyy += Group.sumyy;
    }

    public SqlDouble Terminate()
    {
     return (z * sumxy - x * y) /
            Math.Sqrt(((z * sumxx) - Math.Pow(x, 2))
                   * ((z * sumyy) - Math.Pow(y, 2)));
    }
    private Double x;
    private Double y;
    private Int64 z;
    private Double sumxy;
    private Double sumxx;
    private Double sumyy;
}
```

Dentro de esta UDF agregada cambiamos el tipo de datos al que correspondan nuestros datos entrantes, esto para poder realizar operaciones que sólo se pueden hacer con estos tipos de datos, por ejemplo, para calcular la correlación necesitamos valores numéricos, no podemos hacer cálculos algebraicos con cadenas, por esto realizamos un cambio de tipo en el método ACCUMULATE (de cadenas a valores de tipo Double), y así poder hacer las operaciones correspondientes. Esta UDF agregada se registra en SQL Server de la siguiente manera:

```
 CREATE AGGREGATE corr
```

```
( @o doblentrada )
RETURNS Double
EXTERNAL NAME MiUDA.corr
```

Nuevamente supongamos que el *assembly* llamado *MiUDA* ya fue definido, y contiene el método *corr*. Observemos que el parámetro de entrada para la función de agregación es un valor de tipo *doblentrada*, el cual nos permite establecer como entrada una UDF. Entonces, por último mostramos la forma de invocar esta UDF agregada en SQL Server 2005:

SELECT dbo.corr(dbo.Entbinaria(X, Y)) FROM tablax;

Como podemos observar el parámetro de entrada para la función de agregación es la UDF[15] creada anteriormente, y no tendremos problemas debido a que creamos el tipo *doblentrada*, el cual nos permite realizar esto.

El tipo *doblentrada* y la UDF *Entbinaria* sólo se crean una vez, entonces si deseamos crear una nueva agregación que requiera dos parámetros de entrada, sólo tendremos que crear el código correspondiente a la UDF agregada. También cabe señalar que todo el proceso anterior bien se pudo haber realizado en un solo proyecto en Visual Studio 2005, creando las tres funciones anteriores (UDT, UDF escalar y UDF agregada), compilando el proyecto y finalmente enlazando las funciones con SQL Server 2005 a través de la opción *deploy*. Ahorrando tiempo y esfuerzo al momento de registrar este tipo de funciones.

Para ver otro ejemplo de UDFs agregadas con dos parámetros de entrada véase el apéndice B.

Por último mostramos algunas ventajas y desventajas de utilizar UDFs en los tres SMBDs que estudiamos anteriormente.

[15]Esto debido a que las UDFs escalares si permiten en ingreso de más de un parámetro de entrada.

4.4. Ventajas y desventajas de las UDFs

En conjunción de lo observado anteriormente y basados en [25] relacionamos algunas de las ventajas y desventajas de utilizar las UDFs:

- No hay necesidad de alterar el código interno del manejador.
- Las UDFs son creadas en lenguaje C (en caso de MySQL y PostgreSQL) y una vez compiladas (a código objeto) ellas pueden ser utilizadas en cualquier sentencia "SELECT", como cualquier otra función en SQL. Para el caso de SQL Server 2005, pueden ser construidas mediante $C++$, $C\#$ y Visual Basic .NET.
- El código fuente puede explotar la flexibilidad y rapidez del lenguaje C (MySQL y PostgreSQL).
- Las UDFs se ejecutan en memoria principal, esto es una característica crucial para reducir las E/S a disco y reducir el tiempo de ejecución.
- Las funciones de agregación pueden mantener valores agregados en memoria apilada (heap memory) de fila a fila.
- Actualmente los parámetros de las UDFs en MySQL sólo se pueden definir como tipos de datos simples (String, Integer y Real, pero no arreglos), aunque PostgreSQL y SQL Server 2005 tienen una mayor flexibilidad.
- Las UDFs no pueden realizar operaciones de E/S a disco, esto es, no pueden realizar operaciones del tipo: UPDATE, INSERT ni DELETE.
- Las UDFs sólo pueden devolver datos de tipo simple, en otras palabras, no pueden devolver un conjunto de valores o una matriz, aunque PostgreSQL permite devolver arreglos.
- Las UDFs no pueden internamente invocar la ejecución de otras UDFs.
- El tamaño de las UDFs agregadas en SQL Server 2005 tiene un limite de 8000 bytes.
- El tiempo para construir UDFs esta relacionado con el conocimiento del lenguaje de programación que tenga el usuario.

Capítulo 5

Experimentación

En este capítulo se presenta la evaluación experimental realizada en tres manejadores de bases de datos relacionales: PostgreSQL, MySQL y SQL Server 2005, así como los componentes de la experimentación utilizados, los cuales conforman la alternativa de experimentación propuesta. Las características de los dos servidores sobre los cuales se ejecutaron los experimentos son para el servidor 1: Un Core Duo a 1.66 GHz de velocidad de procesador, disco duro de 120 GB a 5400 RPM y 1 GB de memoria principal, para el servidor 2: 2 procesadores físicos cada uno un Core Duo a 1.6 GHz de velocidad de procesador, disco duro de 160 GB a 15000 RPM y 2 GBs de memoria principal[1]. Los experimentos se realizaron bajo el sistema operativo Linux para PostgreSQL y MySQL, y en el sistema operativo Windows XP para SQL Server 2005. Las versiones de los SMBDs fueron PostgreSQL 8.2.4, MySQL 5.0 y SQL Server 2005. Por otra parte, se realizaron 5 ejecuciones para cada tipo de experimento[2], mismas que se reportaron. El conjunto de datos fue generado a través del programa DBGEN de TPC-H (el cual se explica más adelante), el cual es un generador de datos. Asimismo, se realizaron las pruebas sobre las tablas "lineitem" y "orders"definidas dentro de la DB

[1]Notemos que una técnica de optimización es incrementar las capacidades físicas de nuestro equipo de computo, por esta razón se realizaron los experimentos con dos servidores con distintas características.

[2]Esto debido a que se observó que con este número era suficiente para conocer su comportamiento general.

provista por TPC-H.

¿Qué componente de hardware ayuda más a disminuir el tiempo de ejecución?

En la literatura de bases de datos, se suele recomendar expandir la memoria principal, ya que los algoritmos que realizan las operaciones suelen estar estrechamente relacionados con el tamaño de la memoria RAM; entre mayor cantidad de memoria se tenga entonces un menor número de E/S a disco será requerido, siendo esta operación la mas costosa como se mencionó en los primeros capítulos (en milisegundos y en nano-segundos para la memoria principal). Por esto último se recomienda también tener un disco duro con un RPM alto (≥ 7200) para localizar y extraer los registros de una manera más rápida.

Para mostrar lo anterior se realizó un pequeño experimento para cuantificar el consumo de recursos del sistema, se utilizó el servidor 1 (anteriormente explicado) con 512 Mb y otro con 1Gb de memoria principal y se ejecutaron varias consultas que involucraron funciones de agregación y operaciones binarias (JOIN), esto a efecto de observar si efectivamente la memoria principal es el componente que tiene la prioridad más alta para ser extendido. La base de datos fue la provista por TPC-H con un FE de 1 y 2, se monitoreó el % de CPU y el % de memoria principal utilizados.

Del análisis de los resultados se observó que la memoria principal de 512Mb se ocupó rápidamente y mostró un pico en el % de la memoria utilizada, sugiriendo que requería una mayor cantidad, esto provocó que el % del CPU utilizado disminuyera como consecuencia del efecto de cuello de botella en la memoria principal (al proveer una menor cantidad de registros a ser procesados), este comportamiento también se observó con 1 Gb de memoria principal ya que por la cantidad de RAM requerida por el SO para funcionar la DB no pudo caber completamente en memoria, pero el % del CPU utilizado fue ligeramente mayor, este mismo comportamiento sucedió para el FE 2 donde se observó que la memoria principal se ocupó por completo. Sin embargo, se observó que el plan de ejecución fue más óptimo cuando se tuvo 1 Gb de memoria principal (para ambos FE) ya que permitió utilizar algoritmos más eficientes que cuando sólo se tuvo 512 Mb y esto repercutió directamente

en el uso del disco (incrementándose cuando se tuvo una menor cantidad de memoria principal). Véase el Cuadro 5.1 para observar un ejemplo.

Con este pequeño experimento se mostró que el componente más importante es la memoria principal, ya que permite al optimizador de consultas utilizar algoritmos[3] que requieren una menor cantidad de E/S a disco. Sin embargo, la velocidad del disco y la velocidad del procesador son aspectos que para nada se deben de descuidar, por ejemplo, como se mencionó la velocidad del disco influye a una localización y transferencia de los datos a memoria principal más eficiente, y por ser estas las operaciones más costosas conviene tener una velocidad de disco alta, por otro lado, al tener los datos en memoria principal el procesador será el encargado de realizar los cálculos respectivos y su velocidad también debe ser buena para realizar estas tareas y otras más pertenecientes al sistema de una manera eficiente.

```
                    QUERY PLAN  Con 512 de RAM
-----------------------------------------------------------------------
-----------------------------------------------------------------
 Aggregate (cost=953270.01..953270.02 rows=1 width=13) (actual
time=450813.940..450813.941 rows=1 loops=1)
   ->  Hash Join  (cost=142239.96..923260.18 rows=12003932 width=13)
(actual time=42305.809..393020.687 rows=24004860 loops=1)
        Hash Cond: (t1.l_orderkey = t2.o_orderkey)
        ->  Seq Scan on lineitem t1  (cost=0.00..433564.32 rows=12003932
width=17) (actual time=41.118..237428.436 rows=12002430 loops=1)
        ->  Hash  (cost=93016.65..93016.65 rows=3000265 width=4) (actual
time=30191.022..30191.022 rows=3000000 loops=1)
              ->  Seq Scan on orders t2  (cost=0.00..93016.65
rows=3000265 width=4) (actual time=9.347..24358.601 rows=3000000 loops=1)
 Total runtime: 450814.116 ms

                    QUERY  PLAN   Con 1 Gb de RAM

-----------------------------------------------------------------------
----------------------------------------------------------------------
 Aggregate  (cost=3733175.08..3733175.09 rows=1 width=13) (actual
time=360803.262..360803.263 rows=1 loops=1)
   ->  Merge Join  (cost=3368166.42..3683172.19 rows=20001152 width=13)
(actual time=180218.229..302588.878 rows=24004860 loops=1)
        Merge Cond: (t2.o_orderkey = t1.l_orderkey)
        ->  Sort  (cost=538710.72..546209.12 rows=2999361 width=4)
(actual time=22677.148..28266.860 rows=3000000 loops=1)
              Sort Key: t2.o_orderkey
              ->  Seq Scan on orders t2  (cost=0.00..93007.61
rows=2999361 width=4) (actual time=23.062..13499.887 rows=3000000
loops=1)
        ->  Sort  (cost=2829455.70..2859462.13 rows=12002573 width=17)
(actual time=157518.540..198121.389 rows=24004859 loops=1)
              Sort Key: t1.l_orderkey
              ->  Seq Scan on lineitem t1  (cost=0.00..433550.73
rows=12002573 width=17) (actual time=18.657..81306.669 rows=12002430
loops=1)
 Total runtime: 361259.025 ms
```

Cuadro 5.1: Ejemplos de la planes de ejecución para dos diferentes tamaños de memoria principal y FE de 2.

[3]Recordemos que todos los algoritmos tienen diferentes complejidades las cuales son medidas en E/S a disco.

Por último, sólo resta analizar qué tamaño de memoria principal conviene tener para asegurar que nuestras consultas serán eficientes. Lo ideal sería que el tamaño de la memoria principal fuera mayor que el tamaño de la base de datos, para que el optimizador de consultas pudiera utilizar algoritmos adecuados (quizá hasta de una pasada), sin embargo, las capacidades físicas y económicas actualmente son limitadas, si se tiene una DB muy grande será imposible tener una memoria principal lo suficientemente grande para mejorar el desempeño, entonces sólo resta extender el tamaño hasta donde las especificaciones del ordenador sean permitidas sin perder de vista la velocidad del disco duro y la velocidad del procesador, las cuales serán una grandísima ayuda si su velocidad es alta.

A continuación explicamos los componentes importantes utilizados durante la fase de experimentación. Estos componentes se utilizaron por ser los más ampliamente usados y por tener una estructura general al momento de utilizar las funciones de agregación.

5.1. TPC-H

La importancia de crear una base de datos no sesgada fue importante para nuestras pruebas experimentales, la herramienta que se ha utilizado con buenos resultados para cumplir con esta función en el terreno de las consultas OLAP es TPC-H, la cual permite trabajar con una base de datos y poder hacer pruebas sobre ella. Ésta permite generar distintas bases de datos con diferentes factores de escalación (FE), siendo 100 MB la más pequeña y 100,000 GBs la más grande [31].

Nuestras bases de datos sintéticas fueron generadas por el programa DBGEN de TPC-H, el cual es un generador de datos, con el factor de escalación de 1 y 2 para el servidor 1 y un FE de 1, 2 y 3 para el servidor 2 (por tener un hardware más poderoso). Las tablas definidas dentro de la BD de TPC-H que se utilizaron fueron: “lineitem”la cual contiene aproximadamente 6 millones de registros y “orders” con aproximadamente 1 millon y medio de registros para un FE de 1. El FE de 2 contiene el doble de registros y el FE de 3 el triple.

5.2. Funciones de agregación

En esta sección explicamos las funciones de agregación utilizadas para mostrar la teoría expuesta en los capítulos anteriores.

En el presente trabajo denotaremos a los datos de entrada con las letras X y Y los cuales conforman un vector columna de n datos, esto es, $X = \{x_1, ..., x_n\}$ y $Y = \{y_1, ..., y_n\}$ respectivamente.

Las funciones de agregación se pueden definir según [19] como una clase de funciones genéricas las cuales pueden ser usadas en cualquier aplicación de bases de datos. Los mismos autores clasifican a las funciones de agregación de acuerdo a su complejidad de cómputo. Los tipos más simples de funciones de agregación son:

1. **count** .- Calcula la frecuencia absoluta.
2. **avg**.- Calcula la media aritmética. Se define como: $\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n}$
3. **sum** .- Calcula la suma total.

Los datos de entrada para SUM() y AVG() deben ser una colección de números, sin embargo, otros operadores también pueden operar sobre colecciones de datos de tipo no numérico (ejem. cadenas de caracteres). SQL permite el uso de valores *nulos* para indicar la ausencia de información sobre el valor de un atributo. En general las funciones de agregación ignoran los valores nulos, excepto COUNT(*), la cual para un valor vacío devuelve 0 [1]. Las funciones restantes devuelven un valor nulo cuando se aplican sobre una colección de datos vacía. En el presente trabajo sólo trataremos bases de datos correctas (consistentes y completas) dejando este tipo de problemas para un trabajo futuro.

Las funciones de agregación más complejas son empleadas para hacer estudios más especializados sobre los datos, por ejemplo, para resolver preguntas como: "¿Cuál es la relación entre el atributo A y el atributo B?". Las funciones pertenecientes a esta clase son:

1. **covarianza**.-Mide el grado de relación de dos variables [30]. Se define como:
   $\sigma_{xy} = \frac{1}{n}\sum_{i=1}^{n} x_i y_i - \bar{X}\bar{Y}$

2. **correlación**.-La correlación indica la fuerza y la dirección de una relación lineal entre dos variables aleatorias. La diferencia entre la covarianza y la correlación es que esta última oscila entre el rango de -1 y 1, en cambio la covarianza no esta acotada dentro de un intervalo. Se dirá que existe correlación positiva para el caso de ser cercano a 1, en caso de acercarse a -1 se dirá que tiene una correlación negativa y finalmente en caso de ser cercano a 0 se dirá que no existe correlación lineal entre las variables [30]. El coeficiente de correlación que se empleará será el de Pearson, el cual se calcula de la siguiente manera: $r = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{X}\bar{Y}}{\sqrt{(\sum_{i=1}^{n} x_i^2 - n\overline{X}^2)(\sum_{i=1}^{n} y_i^2 - n\overline{Y}^2)}}$

3. **estimadores de mínimos cuadrados**.-Es una técnica de optimización matemática que, dada una serie de mediciones, encuentra una función que se aproxime a los datos. Minimiza la suma de cuadrados de las diferencias ordenadas entre los puntos generados por la función y los correspondientes a los datos originales. Aunque para el caso de la regresión lineal simple $\widehat{Y} = \beta_0 + \beta_1 x_t$ se reduce a calcular las siguientes ecuaciones [30]:

$$\beta_0 = \bar{Y} - \beta_1 \bar{X} \qquad y \qquad \beta_1 = \frac{\sum_{t=1}^{n} x_t y_t - n\bar{X}\bar{Y}}{\sum_{t=1}^{n} x_t^2 - n\bar{X}^2}$$

Nos enfocaremos al estudio de estas funciones de agregación en UDFs y su equivalente en SQL, para observar las ventajas de utilizar las UDFs agregadas instrumentadas con distintas técnicas de optimización (vistas en los primeros capítulos). Asimismo, analizaremos su desempeño para ambos casos.

Se utilizó este conjunto de funciones debido a que cubren un gran conjunto de posibles casos al momento de construir funciones de agregación

5.3. Tipos de índices utilizados

Los tipos de índices que se implementaron fueron de acuerdo a los soportados por los SMBDs y los más ampliamente utilizados, los cuales se listan a continuación:

- **PostgreSQL**: Este SMBD implementa tres tipos de índices: árboles-B^+, Hash y Gist, sin embargo, sólo implementaremos los árboles-B^+[4].
- **MySQL**: Este SMBD implementa los índices en árboles-B^+.
- **SQL Server 2005**: Este SMBD implementa únicamente los índices en árboles-B^+.

5.4. Organización de los experimentos

El presente estudio está organizado de la siguiente manera: Dividimos las funciones de agregación en dos tipos: (1) Funciones de agregación simples (2) Funciones de agregación complejas. El primer grupo de funciones de agregación comprende las funciones SUMA(), CUENTA() y PROMEDIO(). El segundo grupo contiene las funciones CORRELACION(), COVARIANZA() y REGRESIONL(). Ahora bien, cada tipo de función se evaluó tanto en su versión programada mediante UDFs como en su versión programada en SQL (sin invocar UDFs), pudiéndose invocar funciones intrínsecas del SMBD, cuando estas existan en el SMBD. PostgreSQL en su última versión (8.2.4) cuenta con las funciones intrínsecas CORR(), la cual calcula la correlación lineal entre dos atributos y COVAR_POP(), la cual calcula la covarianza poblacional [16], por lo tanto, en PostgreSQL compararemos en UDFs, en SQL y con la variante invocando la función intrínseca. MySQL y SQL Server 2005 a la fecha no implementan la correlación y la covarianza, por lo cual sólo experimentaremos con la variante en UDFs y la variante en SQL.

[4]Esta decisión fue tomada con base en las observaciones hechas por los desarrolladores de PostgreSQL, donde mencionan que los índices *hash* han mostrado no tener un mejor desempeño que los índices en árboles-B^+.

Para el caso de las funciones de agregación simples se implementarán tanto con la variante en SQL (para este tipo de funciones se entenderá como funciones en SQL a las funciones intrínsecas) como en UDFs utilizando índices (vistos en el capítulo 2) como sin índices, así como con algunas alternativas provistas por las heurísticas del algebra relacional (vistas en la Sección 3.4). Todo esto a efecto de observar si las UDFs agregadas pueden instrumentarse con estas técnicas de optimización y proveer buenos resultados.

Las funciones de agregación complejas sólo serán implementadas en conjunto con y sin índices, esto debido a que en este tipo de funciones generalmente sólo se aplican a una tabla especifica.

Para los experimentos se creó una vez la base de datos por cada operación diferente, y se ejecutaron las consultas cinco veces en forma repetida. Se observó en PostgreSQL (y frecuentemente en los otros dos SMBDs) que a partir de la segunda ejecución el tiempo de respuesta fue menor debido a que en la primera ejecución se realizaba un cálculo automático de estadísticas efectuado por los SMBDs con el objeto de que el optimizador contase dentro del catálogo con esta información para optimizar los planes de ejecución. Por lo anterior, reportaremos los tiempos de todas las repeticiones para poder estudiar este comportamiento.

Por otra parte, las consultas instrumentadas[5] en este trabajo se muestran en el apéndice A. En el lado izquierdo de cada consulta se encuentra un acrónimo, el cual nos ayudará a identificarlas al momento de mostrar sus respectivos resultados de los experimentos. El Cuadro 5.2 resume la lógica utilizada para nombrar las consultas.

Por ejemplo, si tenemos el acrónimo: Ia1.1 se referirá a la función intrínseca AVG() la cual corresponde a la primera consulta del Cuadro A.1 y contiene algún índice. Si tenemos el acrónimo: Uc5 se referirá a la función instrumentada mediante UDFs correspondiente a la función de agregación CUENTA() la cual corresponde a la quinta consulta del Cuadro A.3. Esta misma lógica se debe seguir para el resto de las consultas aquí presentadas.

[5]Estas consultas fueron escogidas a efecto de poder implementar las reglas heurísticas descritas en el capítulo 3 y por tener una estructura general al momento de construir consultas y funciones de agregación.

Letra	Significado
I	Intrínseca
S	SQL
U	UDF
a	promedio
s	suma
c	cuenta
cr	correlación
cv	covarianza
reg	regresión lineal
1,...,7	Identificador de consulta
.1	con índice

Cuadro 5.2: Lógica para los acrónimos de las consultas.

5.5. Resultados

A continuación se muestran los resultados obtenidos[6] por los experimentos explicados anteriormente. Comenzamos con las funciones de agregación simples y finalmente con las funciones de agregación complejas.

5.5.1. Funciones de agregación simples

Empezamos analizando las consultas y los resultados de la función de agregación PROMEDIO(), posteriormente estudiamos las funciones SUMA() y CUENTA() respectivamente.

Resultados de la función de agregación PROMEDIO()

El Cuadro A.1 (véase apéndice) muestra las consultas efectuadas tanto en SQL como en UDFs para la función de agregación PROMEDIO(). Como se mencionó anteriormente, estas consultas son derivadas de las reglas de equivalencia (heurísticas) del algebra relacional (vistas en el capítulo 3), asimismo, se instrumentaron estas consultas con índices, mismas que se identifican con

[6]Sólo se presentara el tiempo de ejecución tomado por cada experimento omitiéndose el plan de ejecución por razones de espacio.

el acrónimo a la que hacen referencia (consulta sin índices) con una terminación de la forma .1.

A continuación se muestran los resultados correspondientes al Cuadro A.1, los cuales fueron ordenados de acuerdo a como aparecen en este Cuadro, los resultados comparan el tiempo de cálculo para realizar la función de agregación PROMEDIO() para los tres SMBDs, tanto con la versión programada mediante UDFs como con la versión en SQL invocando funciones intrínsecas (SQL Intrínseca) [7]. Cada segmento refleja las 5 ejecuciones realizadas (columna n) en los tres SMBDs tanto en UDFs como en SQL para los diferentes factores de escalación (FE), tanto para el servidor 1 como para el servidor 2 (explicados al inicio de este capítulo).

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua1	Ua1.1	Ua1	Ua1.1	Ia1	Ia1.1	Ia1	Ia1.1	Ua1	Ua1.1	Ua1	Ua1.1	Ua1	Ua1.1	Ia1	Ia1.1	Ia1	Ia1.1	Ia1	Ia1.1
PostgreSQL	1	137	69	209	132	135	69	261	130	22	14	78	37	119	57	31	26	99	54	149	80
	2	49	49	99	98	50	49	99	98	14	14	36	38	56	57	25	25	53	53	80	79
	3	48	49	97	98	49	48	98	98	14	14	37	38	57	57	25	25	53	53	80	79
	4	48	48	98	98	49	48	99	97	14	14	37	38	58	57	25	25	53	53	79	79
	5	49	49	97	98	49	48	97	97	14	14	38	38	57	57	25	25	53	53	79	79
promedio		**66**	**53**	**120**	**104**	**66**	**53**	**131**	**104**	**15**	**14**	**45**	**38**	**69**	**57**	**27**	**25**	**62**	**53**	**93**	**79**
MySQL	1	38	77	76	155	42	75	80	157	12	34	25	71	40	114	14	34	27	69	43	118
	2	37	8	73	15	41	8	79	16	9	7	25	14	39	20	10	7	27	14	43	21
	3	37	7	73	16	41	7	78	15	8	7	18	13	39	20	9	7	19	14	42	20
	4	37	7	72	15	41	7	78	15	8	7	18	13	39	20	9	7	19	14	42	20
	5	37	7	72	14	42	7	78	14	8	7	18	13	39	20	9	7	19	13	42	20
promedio		**37**	**21**	**73**	**43**	**41**	**21**	**79**	**44**	**9**	**12**	**21**	**25**	**39**	**39**	**10**	**12**	**22**	**25**	**42**	**40**
SQL SERVER 2005	1	24	4	49	7	24	1	43	2	–	–	–	–	–	–	–	–	–	–	–	–
	2	16	4	31	7	17	1	35	2	–	–	–	–	–	–	–	–	–	–	–	–
	3	15	4	31	7	16	1	32	2	–	–	–	–	–	–	–	–	–	–	–	–
	4	15	4	31	7	16	1	32	2	–	–	–	–	–	–	–	–	–	–	–	–
	5	15	4	31	7	16	1	32	2	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**17**	**4**	**34**	**7**	**18**	**1**	**35**	**2**												

Cuadro 5.3: Tiempo (seg) para calcular las consultas: Ua1, Ua1.1, Ia1 e Ia1.1 pertenecientes al Cuadro A.1 para ambos servidores.

En el Cuadro 5.3 se observa que el tiempo de ejecución es en general más breve en los tres SMBDs para la función de agregación PROMEDIO() en UDFs que su contraparte en SQL (SQL intrínseca). Este comportamiento se observa en ambos servidores[8], aunque para PostgreSQL, el servidor 2 mostró un tiempo

[7] Nos referiremos a funciones intrínsecas en SQL a las funciones que están definidas por defecto en los SMBDs.

[8] No se disponía de una licencia para Windows al momento de realizar los experimentos en el servidor 2, por este motivo no se realizaron los experimentos para SQL Server 2005.

de ejecución mucho menor a favor nuevamente de las UDFs para los tres FE que el mostrado por el servidor 1. Asimismo, podemos observar que al implementar un índice sobre el atributo en el cual se realiza la agregación, generalmente se disminuye el tiempo de ejecución tanto para las UDFs como en SQL, normalmente a favor de las UDFs, excepto en SQL Server 2005 donde el tiempo favoreció a SQL. Podemos observar que la diferencia se hace más notoria conforme el FE aumenta, pero este crecimiento no es lineal ya que influyen algunos factores, como el tamaño de la memoria principal.

Por último podemos observar que el tiempo de ejecución disminuye drásticamente a partir de la segunda ejecución para los tres SMBDs, esto debido a que una vez que se ha realizado una consulta, el SMBD actualiza las estadísticas de la base de datos y esto produce que generalmente el tiempo de ejecución para las siguientes consultas sea menor. Asimismo, observamos que el tiempo de ejecución para el servidor 2 es considerablemente menor que para el servidor 1, esto es obvio debido a que el servidor 2 es mucho más poderoso y esto ayuda gradualmente a tener un tiempo de ejecución mucho menor[9].

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua2	Ua2.1	Ua2	Ua2.1	Ia2	Ia2.1	Ia2	Ia2.1	Ua2	Ua2.1	Ua2	Ua2.1	Ua2	Ua2.1	Ia2	Ia2.1	Ia2	Ia2.1	Ia2	Ia2.1
PostgreSQL	1	129	69	259	131	131	68	262	129	24	19	84	39	128	58	35	31	108	63	163	94
	2	49	49	100	98	50	49	100	97	18	18	39	38	59	57	31	31	63	62	94	94
	3	49	49	97	98	49	49	98	98	18	18	38	38	57	58	30	31	63	63	94	94
	4	48	48	98	98	49	49	97	97	18	18	38	39	58	58	30	31	63	62	94	94
	5	49	49	98	98	48	49	98	98	18	18	38	38	57	58	30	31	63	62	94	94
promedio		**65**	**53**	**130**	**105**	**65**	**52**	**131**	**104**	**19**	**18**	**48**	**39**	**72**	**58**	**31**	**31**	**72**	**63**	**108**	**94**
MySQL	1	223	232	492	505	41	72	80	157	72	62	215	222	349	352	13	36	28	63	50	51
	2	225	226	491	504	42	43	79	87	65	60	218	220	348	353	13	14	29	33	50	50
	3	225	228	481	509	42	42	78	85	65	73	218	218	349	354	13	14	28	33	50	51
	4	225	228	488	506	41	42	78	86	56	61	219	220	350	353	13	14	29	27	50	51
	5	223	229	488	508	41	43	78	86	56	68	218	220	349	354	13	14	28	27	50	51
promedio		**224**	**229**	**488**	**506**	**41**	**49**	**79**	**100**	**63**	**65**	**218**	**220**	**349**	**353**	**13**	**19**	**28**	**37**	**50**	**51**
SQL SERVER 2005	1	36	43	87	82	26	16	42	43	–	–	–	–	–	–	–	–	–	–	–	–
	2	37	36	74	78	19	16	32	33	–	–	–	–	–	–	–	–	–	–	–	–
	3	32	36	74	70	19	16	32	33	–	–	–	–	–	–	–	–	–	–	–	–
	4	33	35	77	74	19	14	35	33	–	–	–	–	–	–	–	–	–	–	–	–
	5	33	35	77	73	17	12	32	32	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**34**	**37**	**78**	**75**	**20**	**15**	**35**	**35**												

Cuadro 5.4: Tiempo (seg) para calcular las consultas: Ua2, Ua2.1, Ia2 e Ia2.1 pertenecientes al Cuadro A.1 para ambos servidores.

[9] Por ejemplo, al tener más memoria principal le permite al optimizador utilizar algoritmos más eficientes.

El Cuadro 5.4 muestra el tiempo de ejecución para la función PROMEDIO() con agrupamiento. Este es un caso interesante dado que PostgreSQL mostró ser en general más eficiente para todos los casos, con y sin índices, con diferentes FE y en ambos servidores. Sin embargo, MySQL y SQL Server 2005 mostraron ser menos eficientes las UDFs para todos los casos, mostrando con esto que estos manejadores no implementan eficientemente las UDFs agregadas que requieran algún agrupamiento. Por último, podemos observar que el utilizar índices generalmente no disminuyo el tiempo de ejecución para estos casos.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua3	Ua3.1	Ua3	Ua3.1	Ia3	Ia3.1	Ia3	Ia3.1	Ua3	Ua3.1	Ua3	Ua3.1	Ua3	Ua3.1	Ia3	Ia3.1	Ia3	Ia3.1	Ia3	Ia3.1
PostgreSQL	1	236	108	597	427	225	122	534	474	61	42	207	148	371	280	73	54	255	201	478	391
	2	199	108	373	427	204	120	421	421	57	24	148	147	279	279	68	36	201	197	390	390
	3	196	107	372	415	201	120	419	420	56	24	147	148	280	279	67	36	196	196	389	389
	4	199	104	373	373	203	119	421	420	55	24	147	147	280	279	66	36	202	196	388	388
	5	197	92	371	372	202	107	420	418	48	24	155	147	279	225	59	36	196	195	375	332
promedio		**206**	**104**	**417**	**403**	**207**	**117**	**443**	**430**	**55**	**27**	**161**	**147**	**298**	**269**	**67**	**40**	**210**	**197**	**404**	**378**
MySQL	1	76	75	347	344	81	85	357	345	39	34	145	144	207	209	42	34	150	151	205	214
	2	73	74	342	344	82	84	355	345	30	30	141	143	205	207	31	30	148	151	217	210
	3	72	73	340	336	81	82	355	340	29	29	140	143	205	202	29	30	148	149	217	210
	4	72	68	340	335	81	75	355	338	29	29	140	140	205	200	30	30	148	147	216	210
	5	72	62	340	340	81	68	352	340	29	30	140	138	199	199	30	30	148	145	216	208
promedio		**73**	**70**	**342**	**340**	**81**	**79**	**355**	**342**	**31**	**30**	**141**	**142**	**204**	**203**	**32**	**31**	**148**	**149**	**214**	**210**
SQL SERVER 2005	1	23	24	74	50	32	30	85	54	–	–	–	–	–	–	–	–	–	–	–	–
	2	22	20	53	39	24	21	65	53	–	–	–	–	–	–	–	–	–	–	–	–
	3	22	20	53	39	24	21	64	56	–	–	–	–	–	–	–	–	–	–	–	–
	4	23	20	55	39	22	21	66	59	–	–	–	–	–	–	–	–	–	–	–	–
	5	22	20	58	39	21	20	63	55	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**22**	**21**	**59**	**41**	**25**	**23**	**69**	**55**												

Cuadro 5.5: Tiempo (seg) para calcular las consultas: Ua3, Ua3.1, Ia3 e Ia3.1 pertenecientes al Cuadro A.1 para ambos servidores.

El Cuadro 5.5 muestra el tiempo de ejecución para la función PROMEDIO() después de aplicar la función JOIN. Aquí podemos observar que para todos los casos las UDFs resultaron ser más eficientes que en SQL para los tres SMBDs, con esto mostramos que las UDFs pueden ser implementadas sobre tablas temporales sin ningún problema. Asimismo, podemos observar que al utilizar índices para este tipo de consultas el tiempo de ejecución se ve disminuido y donde PostgreSQL es el SMBD que se ve más beneficiado al implementarlos.

El Cuadro 5.6 muestra el tiempo de ejecución para la función PROMEDIO() después de aplicar la función JOIN, pero con diferente orden en las relaciones,

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua4	Ua4.1	Ua4	Ua4.1	Ia4	Ia4.1	Ia4	Ia4.1	Ua4	Ua4.1	Ua4	Ua4.1	Ua4	Ua4.1	Ia4	Ia4.1	Ia4	Ia4.1	Ia4	Ia4.1
PostgreSQL	1	255	135	604	369	261	123	653	423	64	42	207	153	370	283	76	55	256	204	480	391
	2	200	108	429	372	205	121	421	422	60	24	156	148	279	280	71	36	201	197	388	390
	3	197	108	429	371	202	118	419	421	57	24	155	147	270	280	69	36	196	196	390	389
	4	198	101	428	371	202	117	422	421	56	24	148	148	280	279	68	36	196	196	390	378
	5	198	94	373	374	202	101	418	417	48	24	148	148	279	225	59	36	195	195	374	332
promedio		**210**	**109**	**452**	**371**	**214**	**116**	**467**	**421**	**57**	**28**	**163**	**149**	**295**	**269**	**68**	**40**	**209**	**197**	**404**	**376**
MySQL	1	73	78	344	351	80	87	351	352	31	31	140	142	206	204	32	32	145	151	214	219
	2	73	77	344	341	80	87	346	339	29	30	140	142	202	204	30	29	144	151	215	210
	3	72	74	344	338	82	87	341	338	29	30	140	140	200	202	30	29	144	147	210	210
	4	73	77	336	335	82	85	341	334	29	30	140	140	200	200	30	30	143	147	213	210
	5	71	76	335	335	81	88	341	332	29	30	140	140	200	200	29	30	144	147	213	209
promedio		**72**	**76**	**341**	**340**	**81**	**87**	**344**	**339**	**29**	**30**	**140**	**141**	**202**	**202**	**30**	**30**	**144**	**149**	**213**	**212**
SQL SERVER 2005	1	21	21	76	59	20	21	84	62	-	-	-	-	-	-	-	-	-	-	-	-
	2	18	20	52	40	19	21	65	58	-	-	-	-	-	-	-	-	-	-	-	-
	3	18	20	52	38	18	21	63	55	-	-	-	-	-	-	-	-	-	-	-	-
	4	18	21	52	39	19	22	64	53	-	-	-	-	-	-	-	-	-	-	-	-
	5	19	21	52	39	18	22	64	52	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**19**	**21**	**57**	**43**	**19**	**21**	**68**	**56**												

Cuadro 5.6: Tiempo (seg) para calcular las consultas: Ua4, Ua4.1, Ia.4 e Ia.4.1 pertenecientes al Cuadro A.1 para ambos servidores.

esto es, después de aplicar una de las reglas de equivalencia (conmutatividad en el JOIN), todo esto a efecto de observar la diferencia en el tiempo de ejecución al intercambiar el orden de las relaciones. Para este caso podemos observar que en general las UDFs fueron más eficientes al obtener resultados semejantes al Cuadro 5.5. Asimismo, podemos observar que conforme el FE aumenta, la diferencia en el desempeño es más notoria a favor de las UDFs en los tres SMBDs. Por otro lado, podemos observar que el instrumentar la regla conmutativa en el orden de las relaciones del JOIN no nos asegura que el tiempo de ejecución sea menor, como en este caso, en donde se muestra que el tiempo de ejecución es muy similar al mostrado en el cuadro anterior, con excepción de las UDFs en PostgreSQL para el servidor 1 y FE de 2, donde se muestra que el tiempo de ejecución si se ve afectado, la razón de esto es que al hacer un cambio en el ordenamiento de las relaciones y poniendo la relación más grande a la izquierda del JOIN obtenemos un plan de ejecución generalmente más óptimo, la Figura 5.1 muestra los planes de ejecución correspondientes a las segundas ejecuciones de los Cuadros 5.5 y 5.6 en donde se observa que en ocasiones el aplicar la regla conmutativa permite al optimizador de consultas utilizar un algoritmo más óptimo lo cual se traduce en un tiempo de ejecución menor. También se observa que el algoritmo

Merge-Join (Ordenación-Mezcla) resultó ser más eficiente que el *Hash-Join*[10] al mostrar un tiempo de respuesta menor. Sin embargo, frecuentemente los optimizadores de consultas son muy eficientes, encontrando en la mayoría de los casos el ordenamiento adecuado para las relaciones.

Relación más grande a la izquierda (Ua3)

```
                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=3733316.76..3733316.77 rows=1 width=13) (actual time=372332.656..372332.657 rows=1 loops=1)
   ->  Merge Join  (cost=3368284.16..3683311.46 rows=20002119 width=13) (actual time=193618.752..315486.722 rows=24004860 loops=1)
         Merge Cond: (t2.o_orderkey = t1.l_orderkey)
         ->  Sort  (cost=538715.96..546214.47 rows=2999403 width=4) (actual time=22055.990..27746.839 rows=3000000 loops=1)
               Sort Key: t2.o_orderkey
               ->  Seq Scan on orders t2  (cost=0.00..93008.03 rows=2999403 width=4) (actual time=12.065..12787.229 rows=3000000 loops=1)
         ->  Sort  (cost=2829568.20..2859575.94 rows=12003096 width=17) (actual time=171562.747..211608.790 rows=24004859 loops=1)
               Sort Key: t1.l_orderkey
               ->  Seq Scan on lineitem t1  (cost=0.00..433555.96 rows=12003096 width=17) (actual time=13.096..83685.180 rows=12002430 loops=1)
 Total runtime: 372553.197 ms
(10 filas)
```

Relación más pequeña a la izquierda (Ua4)

```
                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=953125.15..953125.17 rows=1 width=13) (actual time=428547.322..428547.324 rows=1 loops=1)
   ->  Hash Join  (cost=142253.30..923125.25 rows=11999960 width=13) (actual time=40064.018..371105.174 rows=24004860 loops=1)
         Hash Cond: (t2.l_orderkey = t1.o_orderkey)
         ->  Seq Scan on lineitem t2  (cost=0.00..433524.60 rows=11999960 width=17) (actual time=3.980..235959.820 rows=12002430 loops=1)
         ->  Hash  (cost=93021.69..93021.69 rows=3000769 width=4) (actual time=27877.990..27877.990 rows=3000000 loops=1)
               ->  Seq Scan on orders t1  (cost=0.00..93021.69 rows=3000769 width=4) (actual time=10.415..21889.334 rows=3000000 loops=1)
 Total runtime: 428547.504 ms
(7 filas)
```

Figura 5.1: Planes de ejecución correspondientes a las segundas ejecuciones de las consultas Ua3 y Ua4 en PostgreSQL para el servidor 1 y FE de 2.

El Cuadro 5.7 muestra el tiempo de ejecución para la función PROMEDIO() después de aplicar el operador JOIN, pero ahora implementando la regla distributiva sobre las relaciones para obtener sólo los atributos necesarios para cada relación en el JOIN. Nuevamente podemos observar que para esta alternativa el tiempo de ejecución es menor para las UDFs que en SQL para todos los casos en los tres SMBDs. Asimismo, podemos observar que al instrumentar la regla distributiva en PostgreSQL y en SQL Server 2005 el tiempo de ejecución no se benefició del todo, al contrario, hubo casos en donde el tiempo de ejecución fue peor. Sin embargo, en MySQL esta regla permitió obtener un tiempo de ejecución considerablemente menor. Mostrando con esto que la efectividad de esta regla (y muy probablemente otras más) dependió del SMBDs.

El Cuadro 5.8 muestra el tiempo de ejecución para la función PROMEDIO() después de aplicar el operador JOIN con una condición de búsqueda, ésta es comparada con la misma consulta pero aplicando la regla distributiva para

[10]Se observó que en general el algoritmo más óptimo para este tipo de consultas fue el Merge-Join (para PostgreSQL).

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua5	Ua5.1	Ua5	Ua5.1	Ia5	Ia5.1	Ia5	Ia5.1	Ua5	Ua5.1	Ua5	Ua5.1	Ua5	Ua5.1	Ia5	Ia5.1	Ia5	Ia5.1	Ia5	Ia5.1
PostgreSQL	1	246	134	561	429	226	124	617	416	63	43	208	157	371	282	76	54	255	197	478	391
	2	200	109	427	426	205	120	474	423	59	24	156	149	281	280	70	36	204	195	389	391
	3	198	107	429	423	202	119	422	418	57	24	148	148	284	280	68	36	202	196	391	389
	4	200	100	373	373	204	116	420	420	56	24	147	146	283	279	66	36	196	196	388	380
	5	199	92	373	372	202	100	415	421	49	24	147	147	284	225	59	36	195	196	389	332
promedio		**208**	**109**	**432**	**404**	**208**	**116**	**470**	**419**	**57**	**28**	**161**	**149**	**301**	**269**	**68**	**40**	**210**	**196**	**407**	**377**
MySQL	1	48	64	101	137	53	69	118	137	18	18	42	43	66	70	19	18	45	46	71	70
	2	49	63	102	132	52	69	118	130	17	17	42	42	64	67	17	17	45	44	70	69
	3	48	63	103	133	52	69	117	132	16	17	41	42	64	66	17	17	44	44	70	69
	4	48	64	103	134	52	68	116	130	16	17	41	42	64	66	17	17	44	44	70	68
	5	48	63	104	131	53	69	119	130	16	17	40	43	63	67	17	17	44	44	69	69
promedio		**48**	**64**	**102**	**133**	**53**	**69**	**118**	**132**	**17**	**17**	**41**	**42**	**64**	**67**	**18**	**17**	**44**	**44**	**70**	**69**
SQL SERVER 2005	1	23	21	80	50	23	21	86	62	–	–	–	–	–	–	–	–	–	–	–	–
	2	22	21	62	39	22	21	74	52	–	–	–	–	–	–	–	–	–	–	–	–
	3	21	21	52	40	23	21	71	53	–	–	–	–	–	–	–	–	–	–	–	–
	4	21	20	54	39	23	21	65	51	–	–	–	–	–	–	–	–	–	–	–	–
	5	21	20	54	40	24	20	62	52	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**22**	**21**	**60**	**42**	**23**	**21**	**72**	**54**												

Cuadro 5.7: Tiempo (seg) para calcular las consultas: Ua5, Ua5.1, Ia5 e Ia5.1 pertenecientes al Cuadro A.1 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ua6	Ua7	Ua6	Ua7	Ia6	Ia7	Ia6	Ia7	Ua6	Ua7	Ua6	Ua7	Ua6	Ua7	Ia6	Ia7	Ia6	Ia7	Ia6	Ia7
PostgreSQL	1	181	79	270	162	177	79	416	178	38	14	116	61	180	97	39	15	121	65	190	106
	2	78	83	174	153	80	81	171	169	10	10	61	61	103	97	10	11	65	66	106	103
	3	83	82	162	153	83	84	157	157	9	9	61	62	100	98	11	10	65	65	106	105
	4	82	82	152	154	83	82	158	156	9	9	61	61	98	97	10	10	66	65	106	106
	5	83	83	153	153	82	83	157	158	9	9	60	61	98	97	10	10	65	65	105	101
promedio		**101**	**82**	**182**	**155**	**101**	**82**	**212**	**164**	**15**	**10**	**72**	**61**	**116**	**97**	**16**	**11**	**76**	**65**	**123**	**104**
MySQL	1	73	46	343	110	81	52	349	107	30	12	131	31	191	48	31	12	147	30	203	48
	2	70	45	339	100	79	52	348	108	29	12	131	30	194	47	30	12	147	30	202	48
	3	70	46	337	102	79	52	344	108	29	12	128	31	194	47	30	12	134	31	203	47
	4	59	46	336	102	65	52	340	108	29	12	127	31	193	47	30	12	131	31	203	47
	5	58	46	327	100	63	52	332	108	29	12	127	30	194	47	30	12	130	31	203	47
promedio		**66**	**46**	**337**	**103**	**73**	**52**	**343**	**108**	**29**	**12**	**129**	**31**	**193**	**47**	**30**	**12**	**138**	**31**	**203**	**48**
SQL SERVER 2005	1	32	20	60	76	31	20	72	79	–	–	–	–	–	–	–	–	–	–	–	–
	2	23	19	51	34	24	20	51	40	–	–	–	–	–	–	–	–	–	–	–	–
	3	23	19	51	34	25	20	51	40	–	–	–	–	–	–	–	–	–	–	–	–
	4	24	19	50	34	25	20	51	40	–	–	–	–	–	–	–	–	–	–	–	–
	5	25	19	50	34	25	20	52	40	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**25**	**19**	**52**	**42**	**26**	**20**	**55**	**48**												

Cuadro 5.8: Tiempo (seg) para calcular las consultas: Ua6, Ua7, Ia6 e Ia7 pertenecientes al Cuadro A.1 para ambos servidores.

realizar la condición de búsqueda dentro de la relación correspondiente y además implementando índices. Para este caso podemos observar que generalmente el tiempo de ejecución para las funciones en UDFs y en SQL para el FE de 1 en PostgreSQL es muy similar, no así para factores de escalación más grandes donde el comportamiento se inclino a favor de las UDFs para todos los casos, asimismo, para los dos SMBDs restantes también las UDFs tienen un desempeño mejor que su contraparte en SQL para todos los casos. Por otro lado, podemos observar que el implementar la regla distributiva para calcular la condición de selección antes de realizar la operación JOIN e implementando índices resultó ser una buena alternativa de optimización para MySQL y SQL Server 2005 donde el tiempo de ejecución fue menor. Sin embargo, PostgreSQL no mostró un beneficio al implementar esta alternativa de optimización. Mostrando nuevamente que la efectividad de estas reglas heurísticas están relacionadas con el SMBDs.

Hemos estudiado varias alternativas de utilizar la función de agregación PROMEDIO(): índices, heurísticas del algebra relacional y dos arquitecturas diferentes de hardware. Para esta función resultaron ser generalmente más eficientes las UDFs agregadas que su contraparte en SQL (intrínsecas) para los tres SMBDs.

Resultados de la función de agregación SUMA()

El Cuadro A.2 muestra las consultas realizadas para la función de agregación SUMA(), podemos darnos cuenta de que tienen la misma estructura que las mostradas para la función de agregación PROMEDIO(). A continuación se muestran los resultados devueltos por esta función.

De lo anterior podemos observar que el comportamiento es muy similar al mostrado por la función de agregación PROMEDIO() a favor de las UDFs, excepto PostgreSQL donde mostró un tiempo de ejecución similar al mostrado por SQL.

Observamos que para el Cuadro 5.9 el tiempo de ejecución es en general menor a favor de las UDFs en MySQL y SQL Server 2005 cuando no tienen índices, mostrando que al tenerlos no siempre son más eficientes que en SQL, aunque, podemos observar que generalmente el tiempo se reduce drástica-

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us1	Us1.1	Us1	Us1.1	Is1	Is1.1	Is1	Is1.1	Us1	Us1.1	Us1	Us1.1	Us1	Us1.1	Is1	Is1.1	Is1	Is1.1	Is1	Is1.1
PostgreSQL	1	124	68	211	162	122	68	209	151	22	14	79	37	119	57	24	12	79	36	119	56
	2	47	48	100	133	47	48	99	130	13	13	36	38	56	57	12	12	35	37	56	57
	3	49	48	97	97	48	49	97	102	13	13	37	38	57	57	12	12	37	37	57	57
	4	48	48	98	98	49	48	98	99	13	13	37	38	57	57	12	12	37	38	58	58
	5	49	49	98	97	50	49	98	97	13	13	38	38	57	57	12	12	38	38	57	57
promedio		**63**	**52**	**121**	**118**	**63**	**52**	**120**	**116**	**15**	**13**	**46**	**38**	**69**	**57**	**14**	**12**	**45**	**37**	**69**	**57**
MySQL	1	38	74	76	146	47	77	78	148	13	34	27	71	40	111	13	36	30	73	45	123
	2	38	8	73	14	44	8	76	14	9	7	25	13	39	21	9	7	29	14	42	20
	3	37	7	73	15	43	7	76	14	8	6	18	13	39	20	9	7	19	14	42	20
	4	39	7	72	14	43	7	75	14	8	6	17	13	39	20	9	7	18	14	42	20
	5	38	7	72	15	43	7	75	14	8	6	17	13	38	20	9	7	18	14	43	20
promedio		**38**	**21**	**73**	**41**	**44**	**21**	**76**	**41**	**9**	**12**	**21**	**25**	**39**	**39**	**10**	**13**	**23**	**25**	**43**	**40**
SQL SERVER 2005	1	25	3	42	7	25	2	61	2	-	-	-	-	-	-	-	-	-	-	-	-
	2	17	3	31	6	19	1	36	2	-	-	-	-	-	-	-	-	-	-	-	-
	3	16	3	30	6	19	1	35	2	-	-	-	-	-	-	-	-	-	-	-	-
	4	15	3	30	6	18	1	36	2	-	-	-	-	-	-	-	-	-	-	-	-
	5	13	3	31	6	18	1	36	2	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**17**	**3**	**33**	**6**	**20**	**1**	**41**	**2**												

Cuadro 5.9: Tiempo (seg) para calcular las consultas: Us1, Us1.1, Is1 e Is1.1 pertenecientes al Cuadro A.2 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us2	Us2.1	Us2	Us2.1	Is2	Is2.1	Is2	Is2.1	Us2	Us2.1	Us2	Us2.1	Us2	Us2.1	Is2	Is2.1	Is2	Is2.1	Is2	Is2.1
PostgreSQL	1	133	69	200	130	128	68	209	130	25	17	84	38	128	58	23	16	79	38	122	57
	2	49	48	99	98	50	48	100	97	17	17	39	38	57	57	16	16	38	38	57	58
	3	48	50	98	97	48	49	97	98	17	17	38	38	57	57	16	16	38	38	57	57
	4	49	49	98	97	48	48	97	97	17	17	38	38	58	57	16	16	38	38	57	57
	5	48	49	98	98	48	48	98	98	17	17	38	38	57	58	16	16	38	38	57	57
promedio		**65**	**53**	**118**	**104**	**64**	**53**	**120**	**104**	**19**	**17**	**47**	**38**	**71**	**57**	**18**	**16**	**46**	**38**	**70**	**57**
MySQL	1	221	226	497	509	44	79	80	100	73	79	216	224	351	353	18	35	30	80	55	124
	2	222	227	494	507	44	43	76	86	69	63	219	222	351	353	14	13	27	35	51	50
	3	223	227	493	509	43	43	75	85	66	62	218	223	349	353	14	13	27	34	51	50
	4	224	227	495	511	44	43	76	86	62	56	219	223	350	353	14	13	27	29	51	50
	5	221	227	492	509	44	43	76	86	58	56	220	221	350	353	14	13	27	28	51	50
promedio		**222**	**227**	**494**	**509**	**44**	**50**	**76**	**89**	**66**	**63**	**218**	**223**	**350**	**353**	**15**	**18**	**27**	**41**	**52**	**65**
SQL SERVER 2005	1	38	36	98	74	28	22	58	48	-	-	-	-	-	-	-	-	-	-	-	-
	2	35	35	84	87	23	16	36	41	-	-	-	-	-	-	-	-	-	-	-	-
	3	35	36	76	79	23	16	37	39	-	-	-	-	-	-	-	-	-	-	-	-
	4	34	35	77	72	23	16	34	40	-	-	-	-	-	-	-	-	-	-	-	-
	5	35	44	75	72	23	16	36	40	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**35**	**37**	**82**	**77**	**24**	**17**	**40**	**42**												

Cuadro 5.10: Tiempo (seg) para calcular las consultas: Us2, Us2.1, Is2 e Is2.1 pertenecientes al Cuadro A.2 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us3	Us3.1	Us3	Us3.1	Is3	Is3.1	Is3	Is3.1	Us3	Us3.1	Us3	Us3.1	Us3	Us3.1	Is3	Is3.1	Is3	Is3.1	Is3	Is3.1
PostgreSQL	1	211	130	608	374	247	109	595	424	61	42	199	139	354	262	63	39	206	155	368	277
	2	200	107	369	373	200	108	423	388	58	24	148	139	264	261	59	22	147	148	279	277
	3	198	107	373	373	198	108	423	364	57	24	140	139	262	261	59	22	147	147	277	277
	4	198	101	373	371	196	101	423	363	56	24	139	140	261	261	59	22	147	147	278	268
	5	198	94	374	368	198	83	424	362	49	24	139	139	251	207	48	22	147	145	277	223
promedio		**201**	**108**	**419**	**372**	**208**	**102**	**457**	**380**	**56**	**27**	**153**	**139**	**279**	**250**	**58**	**25**	**159**	**148**	**296**	**264**
MySQL	1	73	74	346	336	81	80	350	341	39	35	140	140	198	205	40	34	144	149	204	208
	2	71	73	344	335	80	80	350	339	30	29	140	140	198	205	30	30	143	148	208	208
	3	72	72	344	332	78	79	345	336	29	29	139	138	196	205	30	30	142	148	208	207
	4	73	66	336	334	80	69	342	335	29	29	139	138	195	202	30	30	143	144	207	204
	5	72	64	332	330	79	69	341	335	29	29	139	138	197	202	30	30	144	144	206	202
promedio		**72**	**70**	**340**	**333**	**79**	**75**	**345**	**337**	**31**	**30**	**139**	**139**	**197**	**204**	**32**	**31**	**143**	**146**	**207**	**206**
SQL SERVER 2005	1	25	26	53	45	29	23	80	48	–	–	–	–	–	–	–	–	–	–	–	–
	2	18	24	54	40	23	18	63	49	–	–	–	–	–	–	–	–	–	–	–	–
	3	18	23	57	40	23	17	64	48	–	–	–	–	–	–	–	–	–	–	–	–
	4	18	23	54	40	23	18	65	49	–	–	–	–	–	–	–	–	–	–	–	–
	5	18	21	55	40	23	18	65	49	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**19**	**24**	**55**	**41**	**24**	**19**	**68**	**49**												

Cuadro 5.11: Tiempo (seg) para calcular las consultas: Us3, Us3.1, Is3 e Is3.1 pertenecientes al Cuadro A.2 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us4	Us4.1	Us4	Us4.1	Is4	Is4.1	Is4	Is4.1	Us4	Us4.1	Us4	Us4.1	Us4	Us4.1	Is4	Is4.1	Is4	Is4.1	Is4	Is4.1
PostgreSQL	1	245	135	566	428	253	130	565	425	63	42	207	147	359	270	62	40	200	147	351	262
	2	201	108	427	371	199	108	424	423	59	24	146	146	271	268	58	22	147	140	261	261
	3	197	107	373	373	199	108	364	363	58	24	147	147	258	268	57	22	140	140	252	261
	4	199	100	369	372	199	100	364	363	56	24	147	147	259	268	55	22	140	139	262	261
	5	198	91	372	373	198	94	363	350	49	24	147	147	257	213	47	22	139	137	252	207
promedio		**208**	**108**	**421**	**383**	**209**	**108**	**416**	**385**	**57**	**27**	**159**	**147**	**281**	**257**	**56**	**26**	**153**	**141**	**276**	**251**
MySQL	1	72	74	332	342	79	83	348	340	34	29	126	141	198	200	35	30	139	148	208	207
	2	73	73	307	341	79	82	340	340	29	29	125	134	198	201	30	30	136	148	208	207
	3	71	74	306	339	80	81	349	339	29	29	124	134	194	200	30	30	132	148	208	207
	4	72	73	304	332	80	83	341	338	29	29	122	134	189	200	30	30	130	148	207	207
	5	72	77	302	331	82	81	347	338	29	29	120	134	194	201	30	30	128	148	205	207
promedio		**72**	**74**	**310**	**337**	**80**	**82**	**345**	**339**	**30**	**29**	**123**	**135**	**195**	**200**	**31**	**30**	**133**	**148**	**207**	**207**
SQL SERVER 2005	1	24	20	53	49	26	31	88	59	–	–	–	–	–	–	–	–	–	–	–	–
	2	17	22	54	40	21	21	63	48	–	–	–	–	–	–	–	–	–	–	–	–
	3	17	21	58	39	20	21	63	47	–	–	–	–	–	–	–	–	–	–	–	–
	4	17	20	57	39	20	14	67	48	–	–	–	–	–	–	–	–	–	–	–	–
	5	17	20	54	40	20	14	66	48	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**18**	**21**	**55**	**41**	**21**	**20**	**69**	**50**												

Cuadro 5.12: Tiempo (seg) para calcular las consultas: Us4, Us4.1, Is4 e Is4.1 pertenecientes al Cuadro A.2 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us5	Us5.1	Us5	Us5.1	Is5	Is5.1	Is5	Is5.1	Us5	Us5.1	Us5	Us5.1	Us5	Us5.1	Is5	Is5.1	Is5	Is5.1	Is5	Is5.1
PostgreSQL	1	247	137	574	363	246	110	565	426	64	42	206	152	363	273	61	40	199	148	353	261
	2	200	108	429	371	203	109	421	363	59	24	155	148	272	273	57	22	147	140	261	261
	3	200	107	371	371	196	108	363	362	57	24	147	147	272	273	57	22	140	140	262	261
	4	197	100	370	371	198	105	362	362	56	24	147	147	273	272	55	22	139	139	261	252
	5	199	92	370	373	196	99	362	357	48	24	146	147	273	268	46	22	139	139	263	257
promedio		**208**	**109**	**423**	**370**	**208**	**106**	**415**	**374**	**57**	**27**	**160**	**148**	**291**	**272**	**55**	**25**	**153**	**141**	**280**	**259**
MySQL	1	48	66	122	130	55	64	122	127	18	18	41	41	67	67	18	18	44	42	69	70
	2	47	65	118	131	55	64	119	131	16	17	41	40	64	66	17	17	44	40	68	68
	3	47	65	118	132	55	64	116	128	17	17	41	41	64	64	17	17	44	40	68	68
	4	48	65	117	130	55	64	117	127	16	17	41	40	64	65	17	17	44	41	68	67
	5	47	66	117	135	56	64	115	127	16	17	40	40	64	65	17	17	43	41	68	68
promedio		**47**	**65**	**119**	**132**	**55**	**64**	**118**	**128**	**17**	**17**	**41**	**40**	**65**	**65**	**17**	**17**	**44**	**41**	**68**	**68**
SQL SERVER 2005	1	22	20	56	46	25	23	86	48	–	–	–	–	–	–	–	–	–	–	–	–
	2	16	20	54	43	20	19	65	47	–	–	–	–	–	–	–	–	–	–	–	–
	3	17	21	54	40	20	19	64	48	–	–	–	–	–	–	–	–	–	–	–	–
	4	16	20	59	40	20	17	61	48	–	–	–	–	–	–	–	–	–	–	–	–
	5	17	20	55	39	20	17	61	49	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**18**	**20**	**55**	**42**	**21**	**19**	**67**	**48**												

Cuadro 5.13: Tiempo (seg) para calcular las consultas: Us5, Us5.1, Is5 e Is5.1 pertenecientes al Cuadro A.2 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Us6	Us7	Us6	Us7	Is6	Is7	Is6	Is7	Us6	Us7	Us6	Us7	Us6	Us7	Is6	Is7	Is6	Is7	Is6	Is7
PostgreSQL	1	204	84	309	160	178	83	290	274	38	14	118	61	179	96	38	13	116	60	179	95
	2	83	82	169	160	83	83	160	162	11	11	60	61	97	97	10	10	60	60	95	96
	3	83	82	166	154	83	82	153	153	10	9	61	61	97	97	10	9	60	62	94	95
	4	82	81	162	153	82	82	152	152	9	9	60	60	96	96	9	9	60	60	94	94
	5	77	79	152	152	79	77	151	151	9	9	59	60	97	96	9	9	60	59	94	94
promedio		**105**	**81**	**191**	**156**	**101**	**81**	**181**	**178**	**15**	**10**	**72**	**61**	**113**	**96**	**15**	**10**	**71**	**60**	**111**	**95**
MySQL	1	73	47	338	108	80	47	350	107	32	12	137	28	194	47	33	12	145	31	201	49
	2	73	47	334	107	78	48	348	106	29	12	132	28	194	47	31	12	143	31	200	50
	3	70	47	332	108	76	47	345	106	29	12	124	29	191	47	31	12	137	31	200	49
	4	62	47	336	108	65	47	340	108	29	12	122	29	194	46	30	12	134	31	199	49
	5	59	47	315	107	65	49	336	107	29	12	122	28	194	47	30	12	129	31	201	49
promedio		**67**	**47**	**331**	**108**	**73**	**47**	**344**	**107**	**30**	**12**	**127**	**28**	**193**	**47**	**31**	**12**	**138**	**31**	**200**	**49**
SQL SERVER 2005	1	22	21	57	81	23	21	70	76	–	–	–	–	–	–	–	–	–	–	–	–
	2	16	23	47	59	17	18	49	51	–	–	–	–	–	–	–	–	–	–	–	–
	3	16	21	47	40	17	17	49	38	–	–	–	–	–	–	–	–	–	–	–	–
	4	16	20	47	39	17	17	50	37	–	–	–	–	–	–	–	–	–	–	–	–
	5	16	20	47	39	17	17	49	38	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**17**	**21**	**49**	**52**	**18**	**18**	**53**	**48**												

Cuadro 5.14: Tiempo (seg) para calcular las consultas: Us6, Us7, Is6 e Is7 pertenecientes al Cuadro A.2 para ambos servidores.

mente al utilizar índices para ambas alternativas. Para PostgreSQL se muestra que el tiempo de ejecución es casi el mismo para ambos casos (UDFs y SQL), así para sus alternativas con índices. Para el Cuadro 5.10 al igual que para la función PROMEDIO(), los agrupamientos son menos eficientes en UDFs para MySQL y SQL Server 2005 con índices y sin ellos, no así para PostgreSQL, donde mostró tener un comportamiento similar para la alternativa en UDFs y en SQL. En el Cuadro 5.11 se observa que en general el tiempo de ejecución es menor a favor de las UDFs para MySQL (con y sin índices) y SQL Server 2005 (sólo sin índices) después de aplicar la operación JOIN. Para PostgreSQL, en los primeros FE se observa un comportamiento muy similar entre las dos alternativas, sin embargo, conforme el FE crece el comportamiento se inclina a favor de las UDFs para todos los casos. Para el Cuadro 5.12 nuevamente podemos observar que para MySQL y SQL Server 2005 los resultados se inclinan generalmente a favor de las UDFs para todos los casos, asimismo, se observa que conforme el FE crece la diferencia a favor de las UDFs aumenta, aunque para sus alternativas con índices este comportamiento no se presenta. Para PostgreSQL se observa que los resultados se inclinan a favor de SQL, aunque de manera marginal. Por otro lado, podemos observar que la regla conmutativa en el ordenamiento de las relaciones en el JOIN no mejoraron mucho los tiempos de ejecución salvo algunos casos en PostgreSQL donde la diferencia fue significativa[11]. Para el Cuadro 5.13 el comportamiento es similar al cuadro anterior, en donde se observa nuevamente que en general las UDFs se desempeñan mejor que en SQL para MySQL y SQL Server 2005, no así para PostgreSQL donde se observa que el tiempo de ejecución es menor para SQL. Por otro lado, podemos observar que la regla distributiva regularmente redujo el tiempo de ejecución para los tres SMBDs, siendo MySQL el más beneficiado al instrumentar esta regla. Finalmente, para el Cuadro 5.14 se observa que para PostgreSQL el tiempo de ejecución es generalmente menor para SQL para ambas alternativas, aunque de manera muy marginal. Para MySQL se observa que generalmente se desempeñan mejor las UDFs para todos los casos. En SQL Server 2005 se observa que para las consultas de la izquierda (Us6 e Is6) el tiempo de ejecución es menor para las UDFs, sin embargo, para las consultas de la derecha (Us7 e Is7) se observa que se invierte este comportamiento. Por otro lado, podemos observar que el instrumentar esta regla distributiva sobre la condición de selección e implementando índices ayudó considerablemente a

[11]Esto debido a los diferentes algoritmos que utilizó el optimizador de consultas.

la disminución en el tiempo de ejecución para los SMBDs MySQL y SQL Server 2005, no así para PostgreSQL donde fuera de la primera ejecución el tiempo de ejecución fue casi el mismo.

Como se observó generalmente el tiempo de ejecución fue menor para las UDFs, asimismo, se observó que la diferencia fue más notoria para MySQL y SQL Server 2005 que para PostgreSQL, donde la diferencia fue en general a favor de las UDFs, aunque, de forma marginal.

Resultados de la función de agregación CUENTA()

Para finalizar con las funciones de agregación simples, pasamos a mostrar los resultados obtenidos para la función CUENTA(). El Cuadro A.3 muestra las consultas realizadas para esta función de agregación, nuevamente la estructura coincide con las anteriores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc1	Uc1.1	Uc1	Uc1.1	Ic1	Ic1.1	Ic1	Ic1.1	Uc1	Uc1.1	Uc1	Uc1.1	Uc1	Uc1.1	Ic1	Ic1.1	Ic1	Ic1.1	Ic1	Ic1.1
PostgreSQL	1	142	69	269	130	137	68	280	132	30	9	79	37	119	57	23	8	82	38	120	56
	2	48	50	99	97	48	49	98	98	13	8	36	37	57	57	13	8	35	38	57	57
	3	49	49	98	99	48	49	98	98	13	8	38	38	57	57	13	8	36	37	57	57
	4	48	48	97	98	48	48	98	97	14	9	37	38	58	57	13	8	36	37	57	58
	5	48	49	97	98	49	49	98	100	13	8	38	38	57	58	13	8	38	36	56	57
promedio		**67**	**53**	**132**	**104**	**66**	**53**	**134**	**105**	**17**	**8**	**46**	**38**	**69**	**57**	**15**	**8**	**45**	**37**	**69**	**57**
MySQL	1	39	74	78	150	40	73	80	152	15	34	27	68	40	114	15	35	28	69	39	115
	2	38	8	73	15	39	8	78	15	9	7	26	13	39	21	10	7	26	13	38	19
	3	39	7	72	15	39	7	79	14	8	7	18	13	39	20	9	6	19	13	38	19
	4	38	7	72	14	38	7	79	14	8	7	17	13	39	20	9	6	18	13	38	19
	5	39	7	72	14	40	7	79	14	8	7	17	13	39	20	9	6	18	13	38	19
promedio		**39**	**21**	**73**	**42**	**39**	**20**	**79**	**42**	**10**	**12**	**21**	**24**	**39**	**39**	**10**	**12**	**22**	**24**	**38**	**38**
SQL SERVER 2005	1	25	4	51	7	29	1	63	3	–	–	–	–	–	–	–	–	–	–	–	–
	2	17	4	30	7	20	1	34	2	–	–	–	–	–	–	–	–	–	–	–	–
	3	16	4	31	7	19	1	34	2	–	–	–	–	–	–	–	–	–	–	–	–
	4	14	4	31	7	18	1	33	2	–	–	–	–	–	–	–	–	–	–	–	–
	5	13	4	31	7	19	1	30	2	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**17**	**4**	**35**	**7**	**21**	**1**	**39**	**2**												

Cuadro 5.15: Tiempo (seg) para calcular las consultas: Uc1, Uc1.1, Ic1 e Ic1.1 pertenecientes al Cuadro A.3 para ambos servidores.

Empezamos con el Cuadro 5.15 en donde el tiempo de ejecución es en general menor a favor de las UDFs para MySQL y SQL Server 2005 cuando éstas no implementan un índice, mostrando que al tenerlo fueron más eficientes

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc2	Uc2.1	Uc2	Uc2.1	Ic2	Ic2.1	Ic2	Ic2.1	Uc2	Uc2.1	Uc2	Uc2.1	Uc2	Uc2.1	Ic2	Ic2.1	Ic2	Ic2.1	Ic2	Ic2.1
PostgreSQL	1	132	69	260	129	125	70	260	130	24	19	84	39	127	57	20	18	79	36	119	56
	2	49	49	98	98	49	49	99	98	18	18	38	38	57	57	18	18	38	37	58	58
	3	49	49	98	98	48	49	98	98	18	18	38	38	58	57	18	18	37	37	57	57
	4	49	49	98	97	48	48	98	98	18	18	38	38	57	57	18	18	37	38	57	57
	5	50	50	98	97	48	49	97	98	18	18	38	38	57	58	19	18	37	38	58	57
promedio		**66**	**53**	**130**	**104**	**64**	**53**	**130**	**104**	**19**	**18**	**47**	**38**	**71**	**57**	**19**	**18**	**46**	**37**	**70**	**57**
MySQL	1	226	233	497	507	38	73	79	150	70	68	217	223	349	353	14	35	27	64	50	51
	2	226	230	493	506	39	43	79	88	73	67	220	220	348	353	14	13	26	34	50	51
	3	226	231	483	506	39	43	78	87	59	59	216	219	349	353	14	13	27	33	50	51
	4	227	230	490	506	39	43	79	88	64	58	218	220	348	354	14	13	27	29	50	51
	5	227	230	491	505	38	43	79	87	54	56	218	220	349	352	14	13	27	28	50	51
promedio		**226**	**231**	**491**	**506**	**38**	**49**	**79**	**100**	**64**	**61**	**218**	**220**	**349**	**353**	**14**	**18**	**27**	**38**	**50**	**51**
SQL SERVER 2005	1	38	45	85	84	28	14	60	50	–	–	–	–	–	–	–	–	–	–	–	–
	2	39	53	79	80	22	14	31	29	–	–	–	–	–	–	–	–	–	–	–	–
	3	35	37	84	72	22	15	31	30	–	–	–	–	–	–	–	–	–	–	–	–
	4	36	36	77	71	22	14	31	29	–	–	–	–	–	–	–	–	–	–	–	–
	5	37	35	80	75	21	14	31	30	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**37**	**41**	**81**	**76**	**23**	**14**	**37**	**34**												

Cuadro 5.16: Tiempo (seg) para calcular las consultas: Uc2, Uc2.1, Ic2 e Ic2.1 pertenecientes al Cuadro A.3 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc3	Uc3.1	Uc3	Uc3.1	Ic3	Ic3.1	Ic3	Ic3.1	Uc3	Uc3.1	Uc3	Uc3.1	Uc3	Uc3.1	Ic3	Ic3.1	Ic3	Ic3.1	Ic3	Ic3.1
PostgreSQL	1	257	135	533	429	250	126	525	421	63	42	184	148	338	251	55	36	184	136	318	231
	2	201	107	371	373	200	109	350	419	57	24	136	153	237	248	53	18	134	134	227	230
	3	200	107	369	373	197	109	350	374	56	24	135	147	248	248	52	18	125	132	220	228
	4	198	99	369	370	197	103	349	351	56	24	135	146	248	238	52	18	125	125	229	228
	5	196	91	369	362	195	93	348	348	49	24	127	147	248	223	45	18	124	124	228	220
promedio		**210**	**108**	**402**	**381**	**208**	**108**	**384**	**383**	**56**	**27**	**143**	**148**	**264**	**242**	**52**	**22**	**138**	**130**	**245**	**227**
MySQL	1	73	74	346	347	80	86	345	353	40	34	139	141	199	200	40	34	147	147	208	206
	2	72	73	338	346	79	85	338	347	31	29	139	141	197	199	31	30	144	146	207	205
	3	73	72	333	345	78	82	333	341	29	29	138	141	199	199	30	30	144	143	205	205
	4	74	64	340	334	79	73	345	338	29	29	138	139	198	199	30	30	141	142	208	205
	5	72	62	339	330	79	69	339	336	29	29	138	132	198	198	30	30	143	137	208	205
promedio		**73**	**69**	**339**	**340**	**79**	**79**	**340**	**343**	**32**	**30**	**138**	**139**	**198**	**199**	**32**	**31**	**144**	**143**	**207**	**205**
SQL SERVER 2005	1	22	27	61	49	34	30	76	49	–	–	–	–	–	–	–	–	–	–	–	–
	2	22	20	51	38	30	23	56	45	–	–	–	–	–	–	–	–	–	–	–	–
	3	22	20	52	39	29	22	55	45	–	–	–	–	–	–	–	–	–	–	–	–
	4	22	20	51	41	29	22	57	45	–	–	–	–	–	–	–	–	–	–	–	–
	5	21	20	51	39	29	21	56	45	–	–	–	–	–	–	–	–	–	–	–	–
promedio		**22**	**21**	**53**	**41**	**30**	**23**	**60**	**46**												

Cuadro 5.17: Tiempo (seg) para calcular las consultas: Uc3, Uc3.1, Ic3 e Ic3.1 pertenecientes al Cuadro A.3 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc4	Uc4.1	Uc4	Uc4.1	Ic4	Ic4.1	Ic4	Ic4.1	Uc4	Uc4.1	Uc4	Uc4.1	Uc4	Uc4.1	Ic4	Ic4.1	Ic4	Ic4.1	Ic4	Ic4.1
PostgreSQL	1	248	135	443	362	209	122	462	400	64	43	187	136	338	248	58	36	184	134	319	231
	2	199	109	426	371	201	109	423	352	59	24	133	131	248	248	55	19	124	125	228	229
	3	200	108	427	370	196	108	422	351	57	24	127	128	248	248	53	18	133	125	218	228
	4	196	99	371	373	197	101	420	350	57	24	127	127	248	248	53	18	124	124	219	228
	5	197	90	370	371	195	94	349	336	48	24	126	121	248	236	45	18	125	123	228	220
promedio		**208**	**108**	**407**	**369**	**200**	**107**	**415**	**358**	**57**	**27**	**140**	**129**	**266**	**245**	**53**	**22**	**138**	**126**	**242**	**227**
MySQL	1	74	73	342	338	78	85	347	346	32	30	138	144	202	200	33	31	144	149	212	205
	2	72	73	338	335	80	85	345	342	29	29	138	141	204	200	30	30	144	148	210	205
	3	74	75	333	331	80	84	339	339	29	29	139	141	204	200	30	30	142	148	210	202
	4	73	74	337	329	79	86	337	334	29	29	138	141	201	201	30	30	142	149	209	200
	5	72	73	337	339	80	82	337	338	29	29	138	140	204	200	30	30	142	148	209	200
promedio		**73**	**74**	**337**	**334**	**79**	**84**	**341**	**340**	**30**	**29**	**138**	**141**	**203**	**200**	**30**	**30**	**143**	**148**	**210**	**202**
SQL SERVER 2005	1	22	20	50	58	34	22	56	55	-	-	-	-	-	-	-	-	-	-	-	-
	2	22	21	51	38	29	22	54	45	-	-	-	-	-	-	-	-	-	-	-	-
	3	22	21	51	38	29	22	54	44	-	-	-	-	-	-	-	-	-	-	-	-
	4	22	21	51	38	29	21	55	45	-	-	-	-	-	-	-	-	-	-	-	-
	5	22	21	52	38	29	21	54	44	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**22**	**21**	**51**	**42**	**30**	**21**	**55**	**47**												

Cuadro 5.18: Tiempo (seg) para calcular las consultas: Uc4, Uc4.1, Ic4 e Ic4.1 pertenecientes al Cuadro A.3 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc5	Uc5.1	Uc5	Uc5.1	Ic5	Ic5.1	Ic5	Ic5.1	Uc5	Uc5.1	Uc5	Uc5.1	Uc5	Uc5.1	Ic5	Ic5.1	Ic5	Ic5.1	Ic5	Ic5.1
PostgreSQL	1	246	138	500	427	240	128	631	420	63	37	187	132	337	250	58	36	185	125	321	230
	2	201	108	372	378	200	109	421	359	58	24	128	131	247	248	57	18	124	132	228	230
	3	198	107	371	372	197	108	349	352	56	24	127	128	248	247	54	18	125	126	230	228
	4	200	99	371	372	197	100	348	349	56	24	126	128	248	247	52	18	125	125	220	228
	5	198	92	373	372	194	91	343	349	50	24	127	132	248	194	50	18	124	125	228	184
promedio		**208**	**109**	**397**	**384**	**206**	**108**	**418**	**366**	**57**	**26**	**139**	**130**	**266**	**237**	**54**	**22**	**137**	**127**	**245**	**220**
MySQL	1	48	63	111	127	55	69	116	134	17	18	41	42	66	71	17	19	45	42	67	70
	2	47	63	115	123	55	68	123	132	16	17	41	40	62	67	17	17	44	40	67	67
	3	48	63	113	125	54	67	120	132	17	17	40	41	62	66	17	17	44	41	67	67
	4	47	64	114	123	54	68	120	132	16	17	40	40	62	66	17	17	44	40	67	67
	5	48	63	115	125	54	68	119	133	16	17	40	41	63	67	17	17	43	41	66	66
promedio		**48**	**63**	**113**	**125**	**54**	**68**	**120**	**132**	**16**	**17**	**40**	**41**	**63**	**67**	**17**	**17**	**44**	**41**	**67**	**67**
SQL SERVER 2005	1	22	20	52	58	28	20	54	54	-	-	-	-	-	-	-	-	-	-	-	-
	2	22	20	51	38	23	20	55	44	-	-	-	-	-	-	-	-	-	-	-	-
	3	22	19	51	38	23	20	55	46	-	-	-	-	-	-	-	-	-	-	-	-
	4	22	19	51	38	23	20	54	46	-	-	-	-	-	-	-	-	-	-	-	-
	5	22	20	52	38	23	20	54	45	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**22**	**20**	**51**	**42**	**24**	**20**	**54**	**47**												

Cuadro 5.19: Tiempo (seg) para calcular las consultas: Uc5, Uc5.1, Ic5 e Ic5.1 pertenecientes al Cuadro A.3 para ambos servidores.

		Servidor 1								Servidor 2											
		UDF				SQLintrinseca				UDF						SQLintrinseca					
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Uc6	Uc7	Uc6	Uc7	Ic6	Ic7	Ic6	Ic7	Uc6	Uc7	Uc6	Uc7	Uc6	Uc7	Ic6	Ic7	Ic6	Ic7	Ic6	Ic7
PostgreSQL	1	196	84	379	153	199	83	303	160	38	14	117	61	180	96	38	13	115	59	176	92
	2	83	82	153	152	84	83	151	151	10	10	62	62	96	96	9	10	59	59	92	89
	3	81	82	152	152	83	82	152	151	10	9	61	63	97	96	9	9	59	59	91	91
	4	80	82	155	152	83	82	151	150	9	9	61	61	97	97	9	9	60	58	90	92
	5	78	78	152	152	79	78	150	150	9	9	61	60	96	97	9	9	59	58	92	92
promedio		**103**	**82**	**198**	**152**	**106**	**81**	**181**	**152**	**15**	**10**	**72**	**61**	**113**	**96**	**15**	**10**	**70**	**58**	**108**	**91**
MySQL	1	73	45	341	103	80	51	347	108	31	12	129	29	194	49	32	12	139	30	202	50
	2	72	46	338	104	79	51	342	109	29	12	128	29	194	48	30	12	138	30	201	49
	3	72	45	335	102	79	51	339	109	29	12	127	29	194	48	30	12	134	31	201	49
	4	60	45	335	102	66	51	324	108	29	12	127	30	194	48	30	12	132	30	201	49
	5	59	45	321	102	65	51	335	109	29	12	125	29	194	48	30	12	131	30	200	49
promedio		**67**	**45**	**334**	**103**	**74**	**51**	**337**	**109**	**29**	**12**	**127**	**29**	**194**	**48**	**31**	**12**	**135**	**30**	**201**	**49**
SQL SERVER 2005	1	21	19	62	84	24	20	67	46	-	-	-	-	-	-	-	-	-	-	-	-
	2	19	19	49	38	18	20	60	36	-	-	-	-	-	-	-	-	-	-	-	-
	3	18	19	49	38	19	20	57	36	-	-	-	-	-	-	-	-	-	-	-	-
	4	18	19	49	37	19	21	47	36	-	-	-	-	-	-	-	-	-	-	-	-
	5	17	19	50	37	19	21	46	36	-	-	-	-	-	-	-	-	-	-	-	-
promedio		**18**	**19**	**52**	**47**	**20**	**21**	**55**	**38**												

Cuadro 5.20: Tiempo (seg) para calcular las consultas: Uc6, Uc7, Ic6 e Ic7 pertenecientes al Cuadro A.3 para ambos servidores.

las funciones en SQL aunque de manera marginal. Para PostgreSQL se observa que el tiempo de ejecución es casi el mismo para ambos casos (UDFs y SQL), así como para sus alternativas con índices. Para el Cuadro 5.16 al igual que para las dos funciones de agregación anteriores, los agrupamientos resultaron ser menos eficientes al instrumentarse mediante UDFs en MySQL y SQL Server 2005 con índices y sin ellos, no así para PostgreSQL, donde mostró tener un comportamiento más estable, mostrando un tiempo de ejecución similar para ambos casos (UDFs y SQL). Para el Cuadro 5.17 se observa que el tiempo de ejecución es menor a favor de las UDFs para MySQL y SQL Server 2005 para todos los casos. En cambio PostgreSQL, para los primeros FE se observa un comportamiento muy similar entre las dos alternativas, sin embargo conforme el FE crece el comportamiento se inclina generalmente a favor de SQL. Para el Cuadro 5.18 podemos observar que para MySQL y SQL Server 2005 los resultados se inclinan a favor de las UDFs para todos los casos. En cambio para PostgreSQL los resultados se inclinan generalmente a favor de SQL. Por otro lado, podemos observar que el implementar la regla conmutativa no introdujo alguna mejora en el tiempo de ejecución, salvo para las UDFs en PostgreSQL donde nuevamente se observa que para el servidor 1 con FE de 2 el tiempo de ejecución es menor cuando la relación más grande esta del lado izquierdo del JOIN. Para el Cuadro 5.19 el comportamiento es

similar al cuadro anterior, en donde se observa que nuevamente las UDFs se desempeñan mejor para MySQL y SQL Server 2005, sin embargo, no así para PostgreSQL donde se observa que el tiempo de ejecución es menor para SQL, aunque sólo para el servidor 2, mientras que para el servidor 1 la diferencia esta a favor de las UDFs para el FE de 2, en cambio para el FE de 1 esta diferencia se inclina a favor de SQL (aunque de manera marginal). Por otro lado, podemos observar que nuevamente el implementar la regla distributiva para obtener sólo los atributos necesarios de las relaciones es una buena alternativa de optimización para MySQL y SQL Server 2005 al mostrar un tiempo generalmente menor al mostrado en los dos cuadros anteriores. Finalmente para el Cuadro 5.20 se observa que para PostgreSQL el tiempo de ejecución es generalmente menor en SQL, aunque de manera muy marginal. Para MySQL se observa que se desempeñan mejor las UDFs en ambos servidores para todos los casos. Por último, para SQL Server 2005 se observa que para la primera alternativa (consultas de la izquierda) el tiempo de ejecución es mejor a favor de las UDFs, no así para la segunda alternativa (consultas de la derecha) donde se observa que para FE de 1 el tiempo de ejecución es menor para las UDFs, en cambio para FE de 2 este comportamiento se invierte. Por otro lado, podemos observar que el instrumentar esta regla distributiva sobre la condición de selección e implementando índices ayudo considerablemente a la disminución en el tiempo de ejecución en MySQL y SQL Server 2005, no así para PostgreSQL donde nuevamente fuera de la primera ejecución el tiempo de ejecución fue casi el mismo.

Como pudimos observar, generalmente el tiempo de ejecución fue menor para las UDFs, se observó nuevamente que la diferencia fue más notoria para MySQL y SQL Server 2005 que para PostgreSQL, este comportamiento se asemeja al mostrado por la función de agregación SUMA(), sugiriendo que tienen un mejor desempeño las funciones de agregación que requieran más de una variable de agregación interna[12] en la UDF (ejem. PROMEDIO()) y tengan un parámetro (atributo) de entrada[13].

De lo anterior podemos observar que en general las UDFs tuvieron un mejor

[12]Con esto nos referimos a la variable que se necesita para agregar los valores de las filas. Por ejemplo, el promedio requiere de dos variables internas, una para mantener la cuenta y otra para la suma.

[13]Esto último sólo para el caso de PostgreSQL, dado que en MySQL ni en SQL Server 2005 se notó este comportamiento.

desempeño que su contraparte en SQL para las funciones de agregación simples. Asimismo, vimos que estas funciones pueden ayudar a disminuir aún más el tiempo de ejecución al instrumentarlas en conjunto con algunas técnicas clásicas de optimización como son: índices, heurísticas del algebra relacional y diferentes arquitecturas de hardware. A continuación se listan las situaciones óptimas para implementar UDFs agregadas simples para los tres SMBDs:

- Para PostgreSQL, se obtiene un mejor desempeño en UDFs donde se requiere más de una variable de agregación interna dentro de la misma UDF (ejem. PROMEDIO()) y un parámetro de entrada en la función de agregación.

- Para MySQL, siempre que no se requiera realizar algún agrupamiento, las UDFs serán generalmente más eficientes.

- Para SQL Server 2005, es similar al punto anterior, mientras no se requiera hacer un agrupamiento, las UDFs se desempeñaran generalmente mejor.

A continuación se muestran los resultados obtenidos para las funciones de agregación complejas, esto es, aquellas funciones que no sólo requieren de más de un parámetro de entrada sino además requieren un cómputo más elaborado para realizar sus cálculos.

5.5.2. Funciones de agregación complejas

Ahora pasamos a estudiar el comportamiento mostrado por las funciones de agregación complejas. Iniciamos nuestro estudio con la función CORRELACION(), posteriormente estudiaremos la función COVARIANZA() y finalmente la función REGRESIONL().

Función de agregación CORRELACION()

Para realizar lo anterior veamos el Cuadro A.4 para observar las consultas realizadas para este tipo de funciones, cabe señalar que sólo se realizaron en su versión simple con y sin índices, debido a que estas funciones normalmente sólo se implementan sobre una tabla especifica.

Para las funciones de agregación complejas se compararon las UDFs con las funciones intrínsecas (sólo para PostgreSQL) y funciones construidas, ambas en SQL.

		Servidor 1											
		UDF				SQLintrinseca				SQL			
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2	
	n	Ucr1	Ucr1.1	Ucr1	Ucr1.1	Icr1	Icr1.1	Icr1	Icr1.1	Scr1	Scr1.1	Scr1	Scr1.1
PostgreSQL	1	136	49	161	98	136	50	193	98	150	55	268	111
	2	49	49	98	98	49	49	98	98	57	56	112	113
	3	49	49	97	98	49	49	99	98	57	56	111	111
	4	50	48	97	98	48	48	98	97	56	56	110	111
	5	49	50	97	98	49	48	98	98	56	56	110	111
promedio		**66**	**49**	**110**	**98**	**66**	**49**	**117**	**98**	**75**	**56**	**142**	**111**
MySQL	1	37	50	91	106	NA	NA	NA	NA	40	78	93	107
	2	37	50	90	105	NA	NA	NA	NA	40	50	92	106
	3	36	49	90	105	NA	NA	NA	NA	40	49	92	107
	4	37	50	91	104	NA	NA	NA	NA	40	50	92	106
	5	36	50	89	105	NA	NA	NA	NA	40	50	93	105
promedio		**37**	**50**	**90**	**105**					**40**	**55**	**92**	**106**
SQL SERVER 2005	1	77	74	162	153	NA	NA	NA	NA	23	26	59	42
	2	75	74	161	151	NA	NA	NA	NA	15	19	34	38
	3	75	74	154	151	NA	NA	NA	NA	15	10	35	33
	4	76	73	152	152	NA	NA	NA	NA	15	10	35	32
	5	76	75	151	151	NA	NA	NA	NA	15	11	34	31
promedio		**76**	**74**	**156**	**152**					**16**	**15**	**39**	**35**

Cuadro 5.21: Tiempo (seg) para calcular la correlación para el servidor 1.

El Cuadro 5.21 compara los resultados para todas las alternativas explicadas anteriormente, aquí podemos observar que para el servidor 1 el desempeño de las UDFs en PostgreSQL es generalmente superior al mostrado por las funciones en su versión intrínseca y construidas en SQL estándar, vemos que las funciones intrínsecas ocupan el segundo lugar en cuanto a eficiencia, seguidas finalmente por las funciones construidas en SQL, mostrándose con esto que las funciones intrínsecas sólo fueron mejoradas por las UDFs. Para el caso donde las UDFs se comparan con la función de correlación construida en SQL, se observa una clara ventaja a favor de las UDFs conforme el FE aumenta para ambas alternativas (con y sin índices). Para MySQL sólo se compararon

		Servidor 2																	
		UDF						SQLintrinseca						SQL					
		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ucr1	Ucr1.1	Ucr1	Ucr1.1	Ucr1	Ucr1.1	Icr1	Icr1.1	Icr1	Icr1.1	Icr1	Icr1.1	Scr1	Scr1.1	Scr1	Scr1.1	Scr1	Scr1.1
PostgreSQL	1	26	17	84	38	124	57	24	17	84	38	128	58	44	40	128	82	192	124
	2	17	17	38	39	58	57	17	17	38	38	64	58	40	40	83	82	124	123
	3	17	17	38	38	57	57	17	17	38	38	57	63	40	40	82	82	124	123
	4	17	17	38	38	58	58	17	17	38	38	58	57	40	40	82	83	123	123
	5	17	17	38	39	57	57	17	17	38	38	57	57	40	40	82	82	124	123
promedio		**19**	**17**	**47**	**38**	**71**	**57**	**19**	**17**	**47**	**38**	**73**	**59**	**41**	**40**	**92**	**82**	**137**	**123**
MySQL	1	14	18	28	34	42	48	NA	NA	NA	NA	NA	NA	15	37	28	59	46	47
	2	10	11	26	32	41	45	NA	NA	NA	NA	NA	NA	12	11	27	32	45	46
	3	10	11	20	32	41	44	NA	NA	NA	NA	NA	NA	11	11	22	32	45	46
	4	10	11	19	31	40	44	NA	NA	NA	NA	NA	NA	11	12	21	31	45	46
	5	10	10	19	31	40	44	NA	NA	NA	NA	NA	NA	11	11	21	27	44	46
promedio		**11**	**12**	**22**	**32**	**41**	**45**							**12**	**16**	**24**	**36**	**45**	**46**

Cuadro 5.22: Tiempo (seg) para calcular la correlación para el servidor 2.

las UDFs con su contraparte construida en SQL[14], aquí se observa que las UDFs se desempeñan mejor para todos los casos (con y sin índices). Por último, para SQL Server 2005 se observó que las funciones construidas en SQL fueron más eficientes que las UDFs. Esto se debe a la pobre implementación que actualmente tiene SQL Server 2005 para construir funciones de agregación que requieran más de un parámetro de entrada. Como se vio en la Sección 4.3.2, tenemos que definir un UDT (User Defined Type) que permita utilizar más de un parámetro, posteriormente se construye una UDF escalar que permita definir una UDF agregada que acepte más de un parámetro de entrada, y finalmente se crea la UDF agregada, la cual tomará como entrada la UDF escalar con los parámetros definidos y el UDT como tipo de dato, y así poder realizar funciones de agregación con más de un parámetro. Entonces como podemos notar, cuando se invoca a la UDF agregada en SQL Server 2005, realmente estamos invocando tres funciones: un UDT, una UDF escalar y una UDF agregada. Por esta razón, el tiempo de ejecución se ve afectado produciendo un tiempo de ejecución mayor para las UDFs en este SMBD[15].

El Cuadro 5.22 correspondiente a la función CORRELACION() para el servidor 2, muestra un comportamiento idéntico al mostrado en el Cuadro 5.21 a

[14]Recordemos que tanto para MySQL y SQL Server 2005 no existen estas funciones de agregación.

[15]En la documentación de SQL Server 2005 se menciona que este problema será resuelto para la versión de SQL Server 2008.

favor de las UDFs para ambos SMBDs. Por último, podemos observar que el implementar índices para la realización de esta función no introdujo ninguna mejora. Al contrario, hay casos en donde el tiempo de ejecución es mayor.

Función de agregación COVARIANZA()

Para esta función de agregación el comportamiento de los resultados fue similar al mostrado por la función CORRELACION(), véase el Cuadro A.4 para observar las consultas realizadas para esta función. Nuevamente se compararon las UDFs contra las funciones intrínsecas (sólo en PostgreSQL) y las construidas en SQL con y sin índices para los tres SMBDs. A continuación se muestran los resultados correspondientes.

		Servidor 1											
		UDF				SQLintrinseca				SQL			
		FE=1		FE=2		FE=1		FE=2		FE=1		FE=2	
	n	Ucv1	Ucv1.1	Ucv1	Ucv1.1	Icv1	Icv1.1	Icv1	Icv1.1	Scv1	Scv1.1	Scv1	Scv1.1
PostgreSQL	1	115	50	197	98	143	49	209	97	139	65	227	130
	2	48	49	99	98	49	48	101	98	67	66	132	130
	3	48	48	97	99	48	48	97	97	66	66	130	131
	4	48	49	97	98	48	48	98	98	66	67	130	130
	5	48	48	98	98	50	48	98	98	66	66	131	131
promedio		**62**	**49**	**118**	**98**	**68**	**48**	**121**	**98**	**81**	**66**	**150**	**131**
MySQL	1	37	50	89	104	NA	NA	NA	NA	40	49	92	106
	2	36	49	88	105	NA	NA	NA	NA	40	50	91	105
	3	37	51	90	105	NA	NA	NA	NA	41	49	92	106
	4	37	50	88	105	NA	NA	NA	NA	39	50	92	106
	5	37	50	89	105	NA	NA	NA	NA	40	49	92	105
promedio		**37**	**50**	**89**	**105**					**40**	**50**	**92**	**106**
SQL SERVER 2005	1	76	74	150	153	NA	NA	NA	NA	19	24	47	36
	2	75	74	150	148	NA	NA	NA	NA	15	17	34	27
	3	76	74	149	147	NA	NA	NA	NA	14	13	26	26
	4	79	74	150	150	NA	NA	NA	NA	14	10	26	26
	5	77	74	149	147	NA	NA	NA	NA	14	10	26	25
promedio		**77**	**74**	**150**	**149**					**15**	**15**	**32**	**28**

Cuadro 5.23: Tiempo (seg) para calcular la covarianza para el servidor 1.

Para el Cuadro 5.23 correspondiente a la función COVARIANZA() para el servidor 1, podemos observar que nuevamente para PostgreSQL las funciones intrínsecas ocupan el segundo lugar en cuanto a eficiencia, seguidas finalmente por las funciones construidas en SQL, mostrándose con esto que las funciones intrínsecas sólo fueron mejoradas por las UDFs. Sin embargo, la diferencia entre las UDFs y las funciones intrínsecas fue de manera marginal al ser implementadas con índices, no así para las funciones construidas en

SQL en donde se destaca una diferencia significativa a favor de las UDFs conforme el FE aumenta. Para MySQL observamos que para la alternativa sin índices, las UDFs se desempeñan mejor en comparación que su contraparte en SQL, por otra parte, para la alternativa con índices se observa un comportamiento similar para ambas funciones (UDFs y SQL), aunque este comportamiento generalmente se inclina a favor de las UDFs conforme el FE aumenta. Por último, para SQL Server 2005 se observa el mismo comportamiento que para la función CORRELACION(), donde el tiempo de ejecución es menor a favor de SQL[16].

		Servidor 2																	
		UDF						SQLintrinseca						SQL					
		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ucv1	Ucv1.1	Ucv1	Ucv1.1	Ucv1	Ucv1.1	Icv1	Icv1.1	Icv1	Icv1.1	Icv1	Icv1.1	Scv1	Scv1.1	Scv1	Scv1.1	Scv1	Scv1.1
PostgreSQL	1	20	17	80	39	112	57	24	17	83	38	124	57	55	52	149	105	224	158
	2	17	17	37	39	57	57	17	17	38	39	57	57	52	52	105	105	158	158
	3	17	17	37	38	57	57	17	17	38	38	57	57	52	52	105	105	157	158
	4	17	17	37	38	57	57	17	17	38	38	58	57	52	52	105	105	158	157
	5	17	17	37	38	56	57	17	17	38	38	57	57	52	52	105	104	157	157
promedio		**18**	**17**	**46**	**38**	**68**	**57**	**19**	**17**	**47**	**38**	**71**	**57**	**52**	**52**	**114**	**105**	**171**	**158**
MySQL	1	10	10	19	21	42	45	NA	NA	NA	NA	NA	NA	12	10	22	21	45	46
	2	10	10	19	19	40	44	NA	NA	NA	NA	NA	NA	11	10	21	20	44	45
	3	10	10	19	19	40	44	NA	NA	NA	NA	NA	NA	11	10	21	20	44	44
	4	10	10	19	19	40	44	NA	NA	NA	NA	NA	NA	11	10	21	20	44	44
	5	10	10	19	19	39	44	NA	NA	NA	NA	NA	NA	11	10	21	20	44	45
promedio		**10**	**10**	**19**	**20**	**40**	**44**							**11**	**10**	**21**	**20**	**44**	**45**

Cuadro 5.24: Tiempo (seg) para calcular la covarianza para el servidor 2.

El Cuadro 5.24 correspondiente a la función COVARIANZA() para el servidor 2 muestra un comportamiento similar al observado en el Cuadro 5.23 a favor de las UDFs para PostgreSQL y MySQL. Por último, podemos observar que el implementar índices para la realización de esta función nuevamente no introdujo ninguna mejora.

Función de agregación REGRESIONL()

Para finalizar las funciones de agregación complejas, se muestra la función de regresión lineal (REGRESIONL()), para esta función sólo se compararon las UDFs contra la función de regresión construida en SQL, esto debido a

[16] Véase la función de CORRELACION() para la explicación respectiva de esta diferencia.

que no existe una función intrínseca para esta función en ninguno de los tres SMBDs, gracias a esto, podemos observar un ejemplo claro en donde la única alternativa que tenemos para construir nuestra función de agregación es mediante SQL estándar o a través de UDFs.

		Servidor 1							
		UDF				SQL			
		FE=1		FE=2		FE=1		FE=2	
	n	Ureg1	Ureg1.1	Ureg1	Ureg1.1	Sreg1	Sreg1.1	Sreg1	Sreg1.1
PostgreSQL	1	118	50	216	98	144	78	303	151
	2	48	49	98	98	75	77	149	148
	3	48	49	98	98	74	78	148	148
	4	48	49	98	98	74	78	148	148
	5	49	49	98	98	74	78	148	149
promedio		**62**	**49**	**122**	**98**	**88**	**78**	**179**	**149**
MySQL	1	36	50	89	105	40	50	92	106
	2	36	49	89	105	40	50	91	105
	3	37	50	89	104	40	50	91	105
	4	37	49	89	105	40	50	91	105
	5	37	50	88	105	40	50	93	106
promedio		**37**	**50**	**89**	**105**	**40**	**50**	**92**	**106**
SQL SERVER 2005	1	75	75	150	150	32	21	36	54
	2	75	74	150	150	16	15	35	36
	3	74	74	149	148	16	12	34	36
	4	74	74	150	150	15	11	35	36
	5	74	74	150	150	16	10	34	37
promedio		**74**	**74**	**150**	**150**	**19**	**14**	**35**	**40**

Cuadro 5.25: Tiempo (seg) para calcular la regresión lineal para el servidor 1.

El Cuadro 5.25 compara los resultados para la función REGRESIONL() devueltos por el servidor 1. Para PostgreSQL podemos observar de manera clara la gran ventaja que tienen las UDFs en relación con la función construida en SQL, además, conforme el FE aumenta esta diferencia en el desempeño aumenta a favor de las UDFs. Por otro lado, para MySQL se observa que las UDFs se desempeñan mejor que su contraparte en SQL, sin embargo, sólo para la alternativa sin índices, mostrando un comportamiento similar para la alternativa con índices. Por último, para SQL Server 2005 se observa un comportamiento similar al mostrado por las funciones de agregación CORRELACION() y COVARIANZA(), donde se muestra un tiempo de ejecución menor para las funciones construidas en SQL (véase la función de CORRELACION() para una explicación).

En el Cuadro 5.26 correspondiente a la función REGRESIONL() para el servidor 2, podemos observar que el tiempo de ejecución es mucho menor

		Servidor 2											
		UDF						SQL					
		FE=1		FE=2		FE=3		FE=1		FE=2		FE=3	
	n	Ureg1	Ureg1.1	Ureg1	Ureg1.1	Ureg1	Ureg1.1	Sreg1	Sreg1.1	Sreg1	Sreg1.1	Sreg1	Sreg1.1
PostgreSQL	1	25	18	84	38	127	57	67	63	167	123	253	185
	2	18	17	38	38	57	57	62	61	124	123	186	185
	3	17	17	38	38	58	58	62	61	124	123	185	185
	4	18	17	38	38	57	57	62	61	123	124	185	185
	5	18	17	38	38	58	58	61	61	123	124	185	185
promedio		**19**	**17**	**47**	**38**	**71**	**57**	**63**	**61**	**132**	**123**	**199**	**185**
MySQL	1	11	11	23	23	46	49	14	15	28	25	52	51
	2	11	11	23	23	45	49	13	13	26	25	50	51
	3	11	11	23	23	45	49	12	12	26	25	50	50
	4	11	11	23	23	45	48	12	13	27	26	50	50
	5	11	11	23	23	44	47	12	13	26	25	49	49
promedio		**11**	**11**	**23**	**23**	**45**	**48**	**13**	**13**	**27**	**25**	**50**	**50**

Cuadro 5.26: Tiempo (seg) para calcular la regresión lineal para el servidor 2.

para las UDFs comparado con el devuelto por SQL en PostgreSQL, asimismo, podemos observar que conforme el FE aumenta esta diferencia se vuelve más notoria siempre a favor de las UDFs para todos los casos, mostrando un comportamiento similar al observado en el Cuadro 5.25.

Por otro lado, para MySQL podemos observar que el desempeño mostrado por las UDFs es superior al mostrado por las funciones construidas en SQL para todos los casos. Sin embargo, la diferencia no fue tan destacada como la mostrada en PostgreSQL. Por último, podemos observar que nuevamente el utilizar índices para este tipo de funciones no mejora el tiempo de ejecución. Mostrando con esto que conviene más utilizar este tipo de funciones sin ningún tipo de índices.

A continuación se muestran las Figuras 5.2 y 5.3 correspondientes al Cuadro 5.26 con la intención de resaltar en forma gráfica las diferencias comentadas para esta función. Estas gráficas muestran claramente las ventajas de las UDFs como una alternativa para la construcción de funciones de agregación complejas en los SMBDs (con excepción de SQL Server 2005).

En esta sección pudimos observar que en general las UDFs tuvieron un mejor desempeño que su contraparte en SQL estándar para las funciones de agregación complejas, excepto para SQL Server 2005 donde se observó un tiempo de ejecución menor para las funciones construidas en SQL, sin embargo, co-

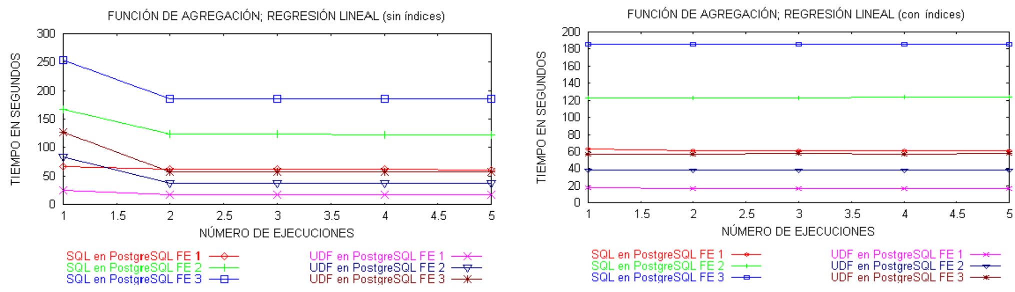

Figura 5.2: Gráfica comparativa de la función de regresión lineal en PostgreSQL con y sin índices para el servidor 2.

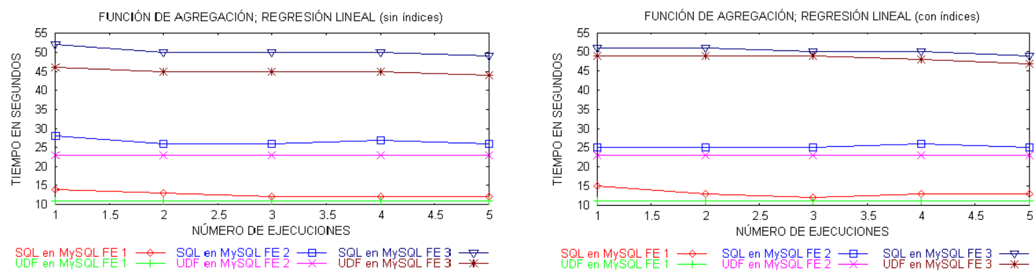

Figura 5.3: Gráfica comparativa de la función de regresión lineal en MySQL con y sin índices para el servidor 2.

mo se explicó anteriormente, esto es debido a la pobre implementación que actualmente tiene SQL Server 2005 para construir funciones de agregación mediante UDFs que requieran más de un parámetro de entrada. Vimos que el SMBD que más se benefició al instrumentar las UDFs para realizar este tipo de funciones (complejas) es PostgreSQL, el cual mostró un tiempo de ejecución considerablemente menor al mostrado por SQL. Por último, si bien en MySQL el tiempo de ejecución es generalmente menor para las UDFs la diferencia fue en general poca en relación con la mostrada por PostgreSQL, sin embargo, sigue siendo una buena alternativa para construir funciones de agregación complejas y de paso evitar el tedioso mantenimiento de código para realizar las funciones de agregación complejas en SQL.

Capítulo 6

Conclusiones

Explicamos el comportamiento de las funciones de agregación implementadas tanto en SQL como en UDFs, en tres SMBDs importantes: PostgreSQL, MySQL y SQL Server 2005. Nos enfocamos en tres funciones de agregación simples: SUMA(), CUENTA() y PROMEDIO(), así como en tres funciones de agregación complejas bien conocidas cuando se desea realizar un análisis estadístico o de minería de datos: CORRELACION(), COVARIANZA() y REGRESIONL(). Para el cálculo de estas funciones de agregación se propusieron tres alternativas: UDFs, funciones intrínsecas en SQL y funciones construidas en SQL (según fuera el caso). Se mostró que en general las UDFs agregadas fueron más eficientes que su contraparte en SQL, aunque se observó que el grado de eficiencia dependió del SMBD. Asimismo, el SMBD que se vio más beneficiado al instrumentar las UDFs agregadas para funciones de agregación complejas fue PostgreSQL, debido a la gran ventaja que mostraron en comparación a las funciones construidas en SQL, también se mostró que para las funciones de agregación simples implementadas mediante UDFs, fueron en general más eficientes que en SQL, aunque de manera marginal, excepto para aquellas que requirieron más de una variable de agregación interna en el código de la UDF, donde se mostró una mayor diferencia a favor de las UDFs. Para MySQL se observó que fueron más eficientes las UDFs agregadas que su contraparte en SQL (excepto para los agrupamientos donde mostró ser el SMBD que peor implementa este tipo de operaciones) tanto para las funciones de agregación simples como para las funciones de

agregación complejas, aunque la diferencia en el desempeño no fue tan significativa como la mostrada en PostgreSQL. En SQL Server 2005 resultaron ser más eficientes (excepto para los agrupamientos) las funciones de agregación simples implementadas mediante UDFs, en general, aquellas funciones de agregación que requirieron de un parámetro de entrada, mientras que se observó un mejor desempeño para las funciones de agregación complejas a favor de SQL, esto debido a la pobre implementación que actualmente tiene SQL Server 2005 para construir funciones de agregación mediante UDFs y que requieran más de un parámetro de entrada. Por otra parte, se observó que los resultados no se vieron afectados por la arquitectura ni por el tamaño de la base de datos, sino que los resultados persistieron (a favor de las UDFs) e incluso se observó que entre más grande la DB más grande era la diferencia a favor de las UDFs (generalmente). Con esto podemos concluir que los resultados son generales.

Sin embargo, para aquellos casos donde no se observó una mejora en el desempeño mediante UDFs, éstas siguen siendo una buena alternativa para construir funciones de agregación y evitar así el tedioso mantenimiento de código para realizar la función de agregación en SQL.

Por otro lado, se mostró que las UDFs no son ajenas a las técnicas de optimización clásica, sino que además, pueden ayudar a tener un desempeño mucho mejor al utilizarlas en su conjunto. Las UDFs se instrumentaron con tres mecanismos de optimización clásica: diferente arquitectura de hardware (expuesto en el Capítulo 2), estructuras de índices (expuestos en el Capítulo 2) y las heurísticas del algebra relacional (expuestas en el Capítulo 3). Todo esto a efecto de mostrar dos de los objetivos de este trabajo: (1) Mostrar que las UDFs agregadas son en general más eficientes que en SQL y (2) extender e incrementar el conocimiento actual sobre el manejo de las UDFs y su aplicación a la teoría de la optimización. Este último se alcanzó al dedicar dos capítulos (correspondientes al 4 y 5) completos para el estudio e implementación de las UDFs para los tres SMBDs (mismos que fueron enriquecidos por la experiencia recabada por este autor), en donde se muestra de manera detallada la metodología para construir tanto UDFs escalares como UDFs agregadas en los tres SMBDs: MySQL, PostgreSQL y SQL Server 2005, así como su instrumentación correspondiente.

Asimismo, se observó que al instrumentar las técnicas de optimización clásica

arriba descritas (índices, reglas heurísticas, etc.) el grado de eficiencia dependió del SMBD y del tipo de técnica aplicada. Por ejemplo, MySQL y SQL Server 2005 mostraron un buen desempeño al instrumentar las reglas heurísticas que disminuyen el número de tuplas y el número de atributos. PostgreSQL y SQL Server 2005 mostraron un mejor desempeño al utilizar los índices en los atributos del JOIN. Sin embargo, la regla conmutativa para el ordenamiento de las relaciones del JOIN generalmente no brindó una mejora sustancial, salvo algunos casos en PostgreSQL donde si redujo el tiempo de ejecución de una manera significativa, para estos casos la relación más grande se localiza en el lado izquierdo del JOIN permitiendo al optimizador de consultas utilizar un algoritmo más eficiente. Esto nos permite inferir que para bases de datos muy grandes conviene definir la relación más grande a la izquierda logrando así que el optimizador utilice un algoritmo más óptimo. Sin embargo, en general los optimizadores de consultas de los tres SMBDs encuentran el ordenamiento adecuado para las relaciones, aunque nunca esta por demás dar una pequeña ayuda al optimizador. Por otro lado, se observó que el utilizar índices para las funciones de agregación complejas no introdujo algún beneficio, dado que el tiempo de ejecución fue similar al mostrado por la versión sin índices. Todo lo anterior en conjunción con el siguiente cuadro deberá ser tomado en cuenta al momento de construir o utilizar alguna función de agregación, esto a efecto de obtener un tiempo de ejecución menor.

A continuación se muestra el Cuadro 6.1, en donde se resumen los hallazgos recabados en este trabajo.

Tipo de función	Caso	PostgreSQL	MySQL	SQL Server 2005
F.A. Simples	Sólo la función de agregación	UDFs*/ INTRIN	UDFs	UDFs
	Agregación con agrupamiento	UDFs*/ INTRIN	SQL/ INTRIN	SQL/ INTRIN
	Agregación después de una operación binaria (ejem. JOIN)	UDFs*/ INTRIN	UDFs	UDFs
	Función de agregación con algún índice	UDFs*/ INTRIN	UDFs	UDFs/ INTRIN
F.A. Complejas	Función de agregación	UDFs/ INTRIN	UDFs	SQL
	Función de agregación con algún índice	UDFs/ INTRIN	UDFs	SQL

*Se obtiene un mejor desempeño cuando las UDFs involucran más de una variable de agregación interna (ejem. Promedio)

Cuadro 6.1: Tabla resumen de la instrumentación de las alternativas estudiadas.

Este cuadro muestra la(s) técnica(s) recomendadas para instrumentar las funciones de agregación, según sea el caso, por ejemplo, supongamos que

queremos construir una función de agregación simple, donde no existe una función intrínseca que realice esta tarea, entonces analizamos el cuadro para observar las demás características que puede contener la función de agregación (índices, agrupamiento, etc), supongamos que sólo queremos construir una función de agregación la cual no requiere ninguna de las características adicionales, esto es, una función de agregación normal que simplemente se aplique a una tabla. Entonces la recomendación en los tres SMBDs sería construirla mediante UDFs o utilizar la función intrínseca si nuestro SMBD fuera PostgreSQL, sin embargo, estamos suponiendo que no existe una función intrínseca que realice esto, por lo tanto, la opción optima sería construirla mediante UDFs. La misma lógica debe seguirse cuando deseemos construir o utilizar (si existe la función intrínseca requerida) funciones de agregación complejas.

En lo que respecta a cuáles son las características de hardware adecuadas para asegurar que nuestras consultas se ejecuten de una manera eficiente, se observó que lo ideal sería que el tamaño de la memoria principal fuera mayor que el tamaño de la base de datos, esto con el fin de que el optimizador de consultas pueda utilizar algoritmos más eficientes (quizá hasta de una pasada), sin embargo, las capacidades físicas y económicas actualmente son limitadas, si se tiene una DB muy grande será imposible tener una memoria principal de igual tamaño, entonces sólo resta extenderla hasta donde las especificaciones del ordenador lo permitan, por otro lado, recordemos que el tamaño de la memoria principal esta relacionada con la elección de los algoritmos que realizan las operaciones, cuyas complejidades están medidas en E/S a disco (operación más costosa), entonces siempre que se pueda incrementar la velocidad del disco duro ($\geq$ 7200 RPM) se estará reduciendo este costo significativamente, independientemente del algoritmo utilizado. Por último, tampoco perdamos de vista la velocidad del procesador (de preferencia un core duo) ya que esto permitirá que el ordenador realice los cálculos respectivos de una manera más rápida.

Por último, la alternativa para la experimentación, la cual consiste en los componentes considerados en la fase de experimentación como son: la base de datos, número de ejecuciones, las reglas heurísticas utilizadas, los FE, los sistemas operativos, las funciones de agregación, los tipos de índices, los equipos de computo, las consultas efectuadas, etc., fue adecuada para poder evaluar el desempeño de instrumentar las funciones de agregación con distin-

tas técnicas y así poder observar cual de éstas resultó ser más eficiente (o bajo que condiciones), asimismo, esta alternativa para la experimentación puede servir como base para extender el estudio de desempeño de las funciones de agregación en otros SMBDs, por ejemplo, mostrar si las UDFs continúan desempeñándose mejor que las funciones en SQL. Esta alternativa para la experimentación puede inclusive ser una referencia valiosa para realizar otro tipo de experimentos relacionados con optimización de SMBDs al considerar todos o alguno de los componentes aquí utilizados.

Apéndice A

Consultas instrumentadas en la fase de experimentación

En esta sección se muestran las consultas efectuadas en la fase de experimentación correspondiente al capítulo 5.

FUNCIÓN DE AGREGACIÓN: AVG ( )

Acrónimo	SQL (avg)	Acrónimo	UDF (promedio)
Ia 1	SELECT avg(l_extendedprice) FROM lineitem;	Ua 1	SELECT promedio(l_extendedprice) FROM lineitem;
Ia 1.1	Ia 1 con índice en l_extendedprice	Ua 1.1	Ua 1 con índice en l_extendedprice
Ia 2	SELECT l_shipmode, avg(l_extendedprice) FROM lineitem GROUP BY l_shipmode;	Ua 2	SELECT l_shipmode, promedio(l_extendedprice) FROM lineitem GROUP BY l_shipmode;
Ia 2.1	Ia 2 con índice en l_extendedprice	Ua 2.1	Ua 2 con índice en l_extendedprice
Ia 3	SELECT avg(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);	Ua 3	SELECT promedio(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);
Ia 3.1	Ia 3 con índices en l_orderkey y o_orderkey	Ua 3.1	Ua 3 con índices en l_orderkey y o_orderkey
Ia 4	SELECT avg(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);	Ua 4	SELECT promedio(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);
Ia 4.1	Ia 4 con índices en l_orderkey y o_orderkey	Ua 4.1	Ua 4 con índices en l_orderkey y o_orderkey
Ia 5	SELECT avg(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);	Ua 5	SELECT promedio(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);
Ia 5.1	Ia 5 con índices en l_orderkey y o_orderkey	Ua 5.1	Ua 5 con índices en l_orderkey y o_orderkey
Ia 6	SELECT avg(t1.l_extendedprice) FROM lineitem as t1 JOIN order; as t2 ON (t1.l_orderkey=t2.o_orderkey, WHERE t1.l_discount= .1 ;	Ua 6	SELECT promedio(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON (t1.l_orderkey=t2.o_orderkey) WHERE t1.l_discount= .1 ;
Ia 7	SELECT avg(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice .OM lineitem WHERE l_discount=.1: as t1 JOIN orders ON (t1.l_orderkey=o_orderkey); Y con índices en l_orderkey y o_orderkey.	Ua 7	SELECT promedio(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice FROM lineitem WHERE l_discount=.1) as t1 JOIN orders ON (t1.l orderkey=o_orderkey); Y con índices en l_orderkey y o_orderkey.

Cuadro A.1: Consultas realizadas para la operación promedio en SQL y en UDFs.

FUNCIÓN DE AGREGACIÓN: SUM( )

Acrónimo	SQL (sum)	Acrónimo	UDF (suma)
ls 1	`SELECT sum( l_extendedprice ) FROM lineitem;`	Us 1	`SELECT suma( l_extendedprice ) FROM lineitem;`
ls 1.1	ls1 con índice en l_extendedprice	Us 1.1	Us 1 con índice en l_extendedprice
ls 2	`SELECT l_shipmode, sum( l_extendedprice) FROM lineitem GROUP BY l_shipmode;`	Us 2	`SELECT l_shipmode, suma( l_extendedprice) FROM lineitem GROUP BY l_shipmode;`
ls 2.1	ls 2 con índice en l_extendedprice	Us 2.1	Us 2 con índice en l_extendedprice
ls 3	`SELECT sum(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);`	Us 3	`SELECT suma(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);`
ls 3.1	ls 3 con índices en l_orderkey y o_orderkey	Us 3.1	Us 3 con índices en l_orderkey y o_orderkey
ls 4	`SELECT sum(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);`	Us 4	`SELECT suma(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);`
ls 4.1	ls 4 con índices en l_orderkey y o_orderkey	Us 4.1	Us 4 con índices en l_orderkey y o_orderkey
ls 5	`SELECT sum(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);`	Us 5	`SELECT suma(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);`
ls 5.1	ls 5 con índices en l_orderkey y o_orderkey	Us 5.1	Us 5 con índices en l_orderkey y o_orderkey
ls 6	`SELECT sum(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON (t1.l_orderkey=t2.o_orderkey) WHERE t1.l_discount= .1 ;`	Us 6	`SELECT suma(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON (t1.l_orderkey=t2.o_orderkey) WHERE t1.l_discount= .1 ;`
ls 7	`SELECT sum(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice FROM lineitem WHERE l_discount=.1) as t1 JOIN orders ON (t1.l_orderkey=o_orderkey);` Y con índices en l_orderkey y o_orderkey.	Us 7	`SELECT suma(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice FROM lineitem WHERE l_discount=.1) as t1 JOIN orders ON (t1.l_orderkey=o_orderkey);` Y con índices en l_orderkey y o_orderkey.

Cuadro A.2: Consultas realizadas para la operación suma en SQL y en UDFs.

FUNCIÓN DE AGREGACIÓN: COUNT ()

Acrónimo	SQL (count)	Acrónimo	UDF (cuenta)
Ic 1	SELECT count(l_extendedprice) FROM lineitem;	Uc 1	SELECT cuenta(l_extendedprice) FROM lineitem;
Ic 1.1	Ic 1 con índice en l_extendedprice	Uc 1.1	Uc 1 con índice en l_extendedprice
Ic 2	SELECT l_shipmode, count(l_extendedprice) FROM lineitem GROUP BY l_shipmode;	Uc 2	SELECT l_shipmode, cuenta(l_extendedprice) FROM lineitem GROUP BY l_shipmode;
Ic 2.1	Ic 2 con índice en l_extendedprice	Uc 2.1	Uc 2 con índice en l_extendedprice
Ic 3	SELECT count(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);	Uc 3	SELECT cuenta(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON(t1.l_orderkey=t2.o_orderkey);
Ic 3.1	Ic 3 con índices en l_orderkey y o_orderkey	Uc 3.1	Uc 3 con índices en l_orderkey y o_orderkey
Ic 4	SELECT count(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);	Uc 4	SELECT cuenta(t2.l_extendedprice) FROM orders as t1 JOIN lineitem as t2 ON(t2.l_orderkey=t1.o_orderkey);
Ic 4.1	Ic 4 con índices en l_orderkey y o_orderkey	Uc 4.1	Uc 4 con índices en l_orderkey y o_orderkey
Ic 5	SELECT count(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);	Uc 5	SELECT cuenta(t1.l_extendedprice) FROM (select l_orderkey, l_extendedprice from lineitem) as t1 JOIN (select o_orderkey from orders) as t2 ON (t1.l_orderkey=t2.o_orderkey);
Ic 5.1	Ic 5 con índices en l_orderkey y o_orderkey	Uc 5.1	Uc 5 con índices en l_orderkey y o_orderkey
Ic 6	SELECT count(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON (t1.l_orderkey=t2.o_orderkey) WHERE t1.l_discount= .1 ;	Uc 6	SELECT cuenta(t1.l_extendedprice) FROM lineitem as t1 JOIN orders as t2 ON (t1.l_orderkey=t2.o_orderkey) WHERE t1.l_discount= .1 ;
Ic 7	SELECT count(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice FROM lineitem WHERE l_discount=.1) as t1 JOIN orders ON (t1.l_orderkey=o_orderkey); Y con índices en l_orderkey y o_orderkey.	Uc 7	SELECT cuenta(t1.l_extendedprice) FROM (SELECT l_orderkey,l_extendedprice FROM lineitem WHERE l_discount=.1) as t1 JOIN orders ON (t1.l_orderkey=o_orderkey); Y con índices en l_orderkey y o_orderkey.

Cuadro A.3: Consultas realizadas para la operación cuenta en SQL y en UDFs.

FUNCIONES DE AGREGACIÓN: CORRELACIÓN ( ), COVARIANZA ( ) y REGRESIÓN LINEAL ( )

Acrónimo	**SQL(correlacion)**	Acrónimo	**SQL intrinsecas (corr)**	Acrónimo	**UDFs (correlacion)**
Scr 1	SELECT (n*Sxy - (Sx*Sy))/sqrt((n*SSx-(Sx*Sx))*(n*SSy- (Sy*Sy))) FROM (SELECT count(*) as n , sum(l_extendedprice) AS Sx, sum(l_quantity) AS Sy, sum(l_extendedprice*l_extendedprice) as SSx, sum(l_quantity*l_quantity) AS SSy, sum(l_quantity*l_extendedprice) AS Sxy FROM lineitem) aggregations ;	Icr 1	SELECT CORR(l_extendedprice, l_quantity) FROM lineitem;	Ucr 1	SELECT correlacion(l_extendedprice, l_quantity) FROM lineitem;
Scr 1.1	Scr 1 con índices en l_extendedprice y l_quantity.	Icr 1.1	Icr 1 con indices en l_extendedprice y l_quantity.	Ucr 1.1	Ucr 1 con índices en l_extendedprice y l_quantity.
	SQL(covarianza)		**SQL intrinsecas (COVAR_POP)**		**UDFs (covarianza)**
Scv 1	SELECT ((prodXY/n)-(mediaX*mediaY)) FROM (select avg(l_extendedprice) AS mediaX, avg(l_quantity) AS mediaY, sum(l_extendedprice*l_quantity) AS prodXY, count(*) AS n FROM lineitem) AS aggregations;	Icv 1	SELECT COVAR_POP(l_extendedprice, l_quantity) FROM lineitem;	Ucv 1	SELECT covarianza(l_extendedprice, l_quantity) FROM lineitem;
Scv 1.1	Scv 1 con índices en l_extendedprice y l_quantity.	Icv 1.1	Icv 1 con índices en l_extendedprice y l_quantity.	Ucv 1.1	Ucv 1 con índices en l_extendedprice y l_quantity.
	SQL(regresion)		**SQL intrinsecas (No aplica)**		**UDFs (regresionL)**
Sreg 1	SELECT (mediaY - (prodXY - n*((mediaX)*(mediaY)))/(cuadX - n*(mediaX*mediaX))*mediaX) AS beta0, (prodXY -n*((mediaX)*(mediaY)))/(cuadX -n*(mediaX*mediaX)) AS beta1 FROM (SELECT sum(l_extendedprice * l_quantity) AS prodXY, avg(l_extendedprice) AS mediaX, avg(l_quantity) AS mediaY, sum(l_extendedprice * l_extendedprice) AS cuadX, count(*) AS n FROM lineitem) AS aggregations;		NA	Ureg 1	SELECT regresionL(l_extendedprice, l_quantity) FROM lineitem;
Sreg 1.1	Sreg 1 con índices en l_extendedprice y l_quantity.		NA	Ureg 1.1	Ureg 1 con índices en l_extendedprice y l_quantity.

Cuadro A.4: Consultas realizadas para la operaciones: correlación, covarianza y regresión lineal en SQL y en UDFs.

Apéndice B

Ejemplos completos de UDFs agregadas

B.1. UDF para la correlación en MySQL

A continuación se muestra el código completo de la función `CORRELACION()` en MySQL.

```
#ifdef STANDARD
/* STANDARD is defined, don't use any mysql functions */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifdef __WIN__
typedef unsigned __int64 ulonglong; /* Microsofts 64 bit types*/
typedef __int64 longlong;
#else
typedef unsigned long long ulonglong;
typedef long long longlong;
#endif /*__WIN__*/
#else
```

```
#include <my_global.h>
#include <my_sys.h>
#if defined(MYSQL_SERVER)
#include <m_string.h>/* To get strmov() */
#else
/* when compiled as standalone */
#define strmov(a,b) strcpy(a,b)
#define bzero(a,b) memset(a,0,b)
#define memcpy_fixed(a,b,c) memcpy(a,b,c)
#endif
#endif
#include <mysql.h>
#include <ctype.h>

#ifdef HAVE_DLOPEN


/* Estas funciones se ponen para evitar que MySQL no encuentre
alguna función*/

my_bool correlation_init( UDF_INIT* initid, UDF_ARGS* args,
 char* message );
void correlation_deinit( UDF_INIT* initid );
void correlation_clear( UDF_INIT* initid, char* is_null,
char *error );
void correlation_add( UDF_INIT* initid, UDF_ARGS* args,
char* is_null, char *error );
double correlation( UDF_INIT* initid, UDF_ARGS* args,
char* is_null, char *error );

struct correlation_data
{
  ulonglong count;
  double totalquantity;
  double totalquantity2;
  double        total;
  double        cuadradox;
  double        cuadradoy;
```

```
};
/*
**Se define la función de inicialización.
*/
my_bool
correlation_init( UDF_INIT* initid, UDF_ARGS* args,
 char* message )
{
  struct correlation_data* data;
  if (args->arg_count != 2)
  {
    strcpy(
   message,
"wrong number of arguments: CORRELATION()
 requires two arguments"
   );
    return 1;
  }
/*
** args->arg_type[0] me representa la primer columna
 que se introduce
*/
  if ((args->arg_type[0] !=  REAL_RESULT) &&
(args->arg_type[0] != INT_RESULT)
 || (args->arg_type[1] != REAL_RESULT)&&
(args->arg_type[1] != INT_RESULT) )
  {
    strcpy(
   message,
   "wrong argument type: CORRELATION()
        requires a REAL or INTEGER"
   );
    return 1;
  }
  /*
 ** Esto fuerza a los argumentos a ser de tipo double.
  */
    args->arg_type[0] = REAL_RESULT;
```

```
    args->arg_type[1] = REAL_RESULT;
  /*
 ** Especificamos ciertas características que deben
 de tener las salidas por ejemplo cuantos decimales
 tendrá la salida.
  */
  initid->maybe_null = 0;
/* El resultado puede ser nulo */
  initid->decimals = 4;
/* Queremos 4 decimales en el resultado */
  initid->max_length = 20;
/* longitud maxima de salida contando a los decimales */
  if (!(data = (struct correlation_data*)
     malloc(sizeof(struct correlation_data))))
  {
    strmov(message,"Couldn't allocate memory");
    return 1;
  }
  data->totalquantity = 0.0;
  data->totalquantity2 = 0.0;
  data->total=0.0;
  data->cuadradox=0.0;
  data->cuadradoy=0.0;
  initid->ptr = (char*)data;
  return 0;
}
  /*
  ** Permite liberar la memoria reservada por la función
   de inicialización.
  */
void
correlation_deinit( UDF_INIT* initid )
{
  free(initid->ptr);
}
  /*
** Resetea las variables a 0 por cada nuevo
agrupamiento.
```

```
  */
void
correlation_clear(UDF_INIT* initid,
char* is_null __attribute__((unused)),
char* message __attribute__((unused)))
{
  struct correlation_data* data =
(struct correlation_data*)initid->ptr;
  data->totalquantity = 0.0;
  data->totalquantity2 = 0.0;
  data->total=0.0;
  data->cuadradox=0.0;
  data->cuadradoy=0.0;
  data->count=0;
}
  /*
  ** Esta función es la que permite tener los valores
agregados, es llamada por cada fila y agrega los valores.
  */
void
correlation_add(UDF_INIT* initid, UDF_ARGS* args,
            char* is_null __attribute__((unused)),
            char* message __attribute__((unused)))
{
  if (args->args[0])
  {
    struct correlation_data* data =
   (struct correlation_data*)initid->ptr;
    double quantity = *((double*)args->args[0]);
double newquantity = data->totalquantity + quantity;
double quantity2 = *((double*)args->args[1]);
double newquantity2 = data->totalquantity2 + quantity2;
double newtotal     = data->totalquantity + quantity;
double newcuadradox = data->totalquantity + quantity;
double newcuadradoy = data->totalquantity2 + quantity2;

    data->count++ ;
    data->totalquantity  += quantity;
```

```
    data->totalquantity2 += quantity2;
    data->total           += quantity * quantity2;
    data->cuadradox       += pow(quantity,2);
    data->cuadradoy       += pow(quantity2,2);
}
  }
  /*
  ** Esta es la función principal que devuelve
        los resultados finales.
  */
double
correlation( UDF_INIT* initid,
    UDF_ARGS* args __attribute__((unused)),
    char* is_null, char* error __attribute__((unused)))
{
  struct correlation_data* data =
 (struct correlation_data*)initid->ptr;
  if (!data->count || !data->totalquantity ||
                                 !data->totalquantity2)
  {
    *is_null = 1;
    return 0.0;
  }
  *is_null = 0;
  return ((data->count)* (data->total) -
 (data->totalquantity)*(data->totalquantity2))/pow(
 (((data->count)*(data->cuadradox) -
 pow((data->totalquantity),2))*((data->count)*
 (data->cuadradoy) - pow((data->totalquantity2),2))),0.5);
}

#endif /* HAVE_DLOPEN */
```

B.2. UDF para la suma en PostgreSQL

A continuación se muestra el código completo de la función `SUMA()` en PostgreSQL.

```
#include "postgres.h"
#include <ctype.h>
#include <limits.h>
#include <math.h>
#include "funcapi.h"
#include "libpq/pqformat.h"
#include "nodes/nodes.h"
#include "utils/int8.h"
#define MAXINT8LEN      25
#define SAMESIGN(a,b)   (((a) < 0) == ((b) < 0))
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
PG_FUNCTION_INFO_V1(suma);
Datum
 suma(PG_FUNCTION_ARGS)
 {
     float8       newval;
     if (PG_ARGISNULL(0))
     {
          if (PG_ARGISNULL(1))
             PG_RETURN_NULL();
         newval = (float8) PG_GETARG_FLOAT4(1);
         PG_RETURN_FLOAT8(newval);
     }
     if (fcinfo->context && IsA(fcinfo->context, AggState))
     {
        float8      *oldsum = (float8 *) PG_GETARG_POINTER(0);
 /* Leave the running sum unchanged in the new input is null */
         if (!PG_ARGISNULL(1))
            *oldsum = *oldsum + (float8) PG_GETARG_FLOAT4(1);
         PG_RETURN_POINTER(oldsum);
```

```
    }
    else
    {
        float8       oldsum = PG_GETARG_FLOAT4(0);
        /* Leave sum unchanged if new input is null. */
        if (PG_ARGISNULL(1))
            PG_RETURN_FLOAT8(oldsum);
        /* OK to do the addition. */
        newval = oldsum + (float8) PG_GETARG_FLOAT4(1);
        PG_RETURN_FLOAT8(newval);
    }
}
```

B.3. UDF para la correlación en PostgreSQL

A continuación se muestra el código completo de la función `CORRELACION()` en PostgreSQL.

```
#include "postgres.h"
#include <ctype.h>
#include <float.h>
#include <math.h>
#include <limits.h>
#include "catalog/pg_type.h"
#include "libpq/pqformat.h"
#include "utils/array.h"
#include "utils/builtins.h"
#if defined(WIN32) && !defined(NAN)
static const uint32 nan[2] = {0xffffffff, 0x7fffffff};
#define NAN (*(const double *) nan)
#endif
/*
* check to see if a float4/8 val
*/
```

```
#define CHECKFLOATVAL(val, inf_is_valid, zero_is_valid)      \
do {                                                         \
    if (isinf(val) && !(inf_is_valid))                       \
       ereport(ERROR,                                        \
                (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE), \
         errmsg("value out of range: overflow")));           \
                                                             \
   if ((val) == 0.0 && !(zero_is_valid))                     \
        ereport(ERROR,                                       \
                (errcode(ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE),\
         errmsg("value out of range: underflow")));           \
 } while(0)
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
PG_FUNCTION_INFO_V1(check_float8_array);
static float8 *
check_float8_array(ArrayType *transarray,
                                  const char *caller, int n)
 {
    /*
* We expect the input to be an N-element float array;
* verify that. We
* don't need to use deconstruct_array() since
* the array data is just
* going to look like a C array of N float8 values.
   */
   if (ARR_NDIM(transarray) != 1  ||
        //ARR_DIMS(transarray)[0] != n )||
       //ARR_HASNULL(transarray) ||
         ARR_ELEMTYPE(transarray) != FLOAT8OID)
elog(ERROR, "%s: expected %d-element float8 array", caller, n);
    return (float8 *) ARR_DATA_PTR(transarray);
 }
PG_FUNCTION_INFO_V1(acumulacorr);
 Datum
 acumulacorr(PG_FUNCTION_ARGS)
 {
```

```
     ArrayType  *transarray = PG_GETARG_ARRAYTYPE_P(0);
     float8      newvalY = PG_GETARG_FLOAT4(1);
     float8      newvalX = PG_GETARG_FLOAT4(2);
     float8     *transvalues;
     float8      N,
                 sumX,
                 sumX2,
                 sumY,
                sumY2,
                 sumXY;
transvalues = check_float8_array(transarray, "acumulacorr", 6);
     N = transvalues[0];
     sumX = transvalues[1];
     sumX2 = transvalues[2];
     sumY = transvalues[3];
     sumY2 = transvalues[4];
     sumXY = transvalues[5];
     N += 1.0;
 sumX += newvalX;
 CHECKFLOATVAL(sumX, isinf(transvalues[1])
                                  || isinf(newvalX), true);
 sumX2 += newvalX * newvalX;
 CHECKFLOATVAL(sumX2, isinf(transvalues[2])
                                  || isinf(newvalX), true);
 sumY += newvalY;
 CHECKFLOATVAL(sumY, isinf(transvalues[3])
                                  || isinf(newvalY), true);
 sumY2 += newvalY * newvalY;
 CHECKFLOATVAL(sumY2, isinf(transvalues[4])
                                  || isinf(newvalY), true);
 sumXY += newvalX * newvalY;
 CHECKFLOATVAL(sumXY, isinf(transvalues[5])
                                  || isinf(newvalX) ||
                 isinf(newvalY), true);

     if (fcinfo->context && IsA(fcinfo->context, AggState))
     {
         transvalues[0] = N;
```

```
        transvalues[1] = sumX;
        transvalues[2] = sumX2;
        transvalues[3] = sumY;
        transvalues[4] = sumY2;
        transvalues[5] = sumXY;

        PG_RETURN_ARRAYTYPE_P(transarray);
    }
    else
    {
        Datum       transdatums[6];
        ArrayType  *result;

        transdatums[0] = Float8GetDatumFast(N);
        transdatums[1] = Float8GetDatumFast(sumX);
        transdatums[2] = Float8GetDatumFast(sumX2);
        transdatums[3] = Float8GetDatumFast(sumY);
        transdatums[4] = Float8GetDatumFast(sumY2);
        transdatums[5] = Float8GetDatumFast(sumXY);

        result = construct_array(transdatums, 6,
                                 FLOAT8OID,
                                 sizeof(float8),
                  false /* float8 byval */ , 'd');
        PG_RETURN_ARRAYTYPE_P(result);
    }
 }

PG_FUNCTION_INFO_V1(fcorr);
Datum
 fcorr(PG_FUNCTION_ARGS)
 {
     ArrayType  *transarray = PG_GETARG_ARRAYTYPE_P(0);
     float8     *transvalues;
     float8      N,
                 sumX,
                 sumX2,
                 sumY,
```

```
                sumY2,
                sumXY,
                numeratorX,
                numeratorY,
                numeratorXY;

transvalues = check_float8_array(transarray, "fcorr", 6);
    N = transvalues[0];
    sumX = transvalues[1];
    sumX2 = transvalues[2];
    sumY = transvalues[3];
    sumY2 = transvalues[4];
    sumXY = transvalues[5];

    /* if N is 0 we should return NULL */
    if (N < 1.0)
        PG_RETURN_NULL();

numeratorX = N * sumX2 - sumX * sumX;
CHECKFLOATVAL(numeratorX, isinf(sumX2) || isinf(sumX), true);
numeratorY = N * sumY2 - sumY * sumY;
CHECKFLOATVAL(numeratorY, isinf(sumY2) || isinf(sumY), true);
numeratorXY = N * sumXY - sumX * sumY;
CHECKFLOATVAL(numeratorXY, isinf(sumXY) || isinf(sumX) ||
                   isinf(sumY), true);
   if (numeratorX <= 0 || numeratorY <= 0)
       PG_RETURN_NULL();

    PG_RETURN_FLOAT8(sqrt((numeratorXY * numeratorXY) /
                          (numeratorX * numeratorY)));
}
```

B.4. UDF para la regresión lineal en SQL Server 2005

A continuación se muestra el código de la función `REGRESIONL()`, aunque sólo mostramos el correspondiente a la UDF agregada, la definición del UDT y la UDF escalar son los mismos que mostramos en la sección 4.3.2.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.
                  SqlUserDefinedAggregate(Format.Native)]
public struct regresionL
{
    public void Init()
    {
        x = 0.0;
        y = 0.0;
        z = 0;
        sumxy = 0.0;
        sumxx = 0.0;
    }
    public void Accumulate(doblentrada o)
    {
        Double var1 = Double.Parse(o.primera);
        Double var2 = Double.Parse(o.segunda);
        x += var1;
        y += var2;
        z += 1;
        sumxx += Math.Pow(var1, 2);
        sumxy += (var1) * (var2);
   }
```

```
    public void Merge(regresionL Group)
    {
        x += Group.x;
        y += Group.y;
        z += Group.z;
        sumxx += Group.sumxx;
        sumxy += Group.sumxy;
    }
    public SqlString Terminate()
    {
  SqlString beta1 = Convert.ToString((sumxy - z * ((x / z)
          * (y / z))) / (sumxx - z * Math.Pow(x / z, 2)));

  SqlString beta0 = Convert.ToString((y / z) - ((sumxy - z *
     ((x / z)* (y / z))) / (sumxx - z * Math.Pow(x / z, 2)))
                                                 * (x / z));
        return (" Bo  " + beta0 + " , B1  " + beta1);
    }
    private Double x;
    private Double y;
    private Int64 z;
    private Double sumxy;
    private Double sumxx;
}
```

Cabe señalar que esta función se debe definir como:

SELECT dbo.regresionL(dbo.Entbinaria(Y,X)) FROM table;

Donde Y es la variable dependiente y X es la variable independiente.

Bibliografía

[1] Henry F. Korth Abraham Silberschatz and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, fifth edition, 2005.

[2] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] Haran Boral and David J DeWitt. A methodology for database system performance evaluation. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 176–185, New York, NY, USA, 1984. ACM.

[5] Surajit Chaudhuri. An overview of query optimization in relational systems. In *ACM PODS '98*, pages 34–43, New York, NY, USA, 1998.

[6] Date C.J. *An Introduction to Database Systems*. Addison Wesley, 2003.

[7] Sara Cohen. User-defined aggregate functions: bridging theory and practice. In *ACM SIGMOD '06*, pages 49–60, New York, NY, USA, 2006.

[8] Derek Comingore and Douglas Hinson. *Professional SQL Server 2005 CLR Programming*. Wrox, 2006.

[9] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD '84*, pages 1–8, New York, NY, USA, 1984.

[10] Robin Dewson and Julian Skinner. *Pro SQL Server assemblies*. Apress, 2005.

[11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, 2006.

[12] Jane Fedorowicz. Database performance evaluation in an indexed file environment. *ACM Trans. Database Syst.*, pages 85–110, 1987.

[13] Leo R. Gotlieb. Computing joins of relations. In *ACM SIGMOD '75*, pages 55–63, New York, NY, USA, 1975.

[14] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, pages 73–170, 1993.

[15] MySQL Global Development Group. Mysql 5.1 manual reference. `http://dev.mysql.com/doc/refman/5.1/en/`.

[16] PostgreSQL Global Development Group. Postgresql 8.2.3: Sql conformance. `http://www.postgresql.org/docs/8.2/interactive/index.html`.

[17] Jeffrey D. Ullman Hector Garcia-Molina and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[18] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[19] H. J. Lenz and B. Thalheim. OLAP databases and aggregation functions. In *SSDM*, pages 91–100, 2001.

[20] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *IEEE Computer Society '80*, pages 212–223, 1980.

[21] David B. Lomet. Bounded index exponential hashing. *ACM Trans. Database Syst.*, 8(1):136–165, 1983.

[22] Christian Mena-Ruvalcaba and Javier García-García. Estudio de desempeño de funciones de agregación utilizando funciones definidas por el usuario (udfs) en postgresql y mysql. In *Primer Encuentro de Estudiantes en Ciencias de la Computación E2C2 '07*, pages 1–10, México, DF, 2007.

[23] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *ACM SIGMOD '97*, pages 38–49, New York, NY, USA, 1997.

[24] C. Ordonez and J. García-García. Consistent aggregations in databases with referential integrity errors. In *ACM IQIS Workshop*, 2006.

[25] Carlos Ordonez. Building statistical models and scoring with udfs. In *ACM SIGMOD Conference*, pages 1005–1016, 2007.

[26] Carlos Ordonez and Javier García-García. Vector and matrix operations programmed with udfs in a relational dbms. In *ACM CIKM '06*, pages 503–512, New York, NY, USA, 2006.

[27] Carlos Ordonez and Javier García-García. Vector and matrix operations programmed with udfs in a relational dbms. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 503–512, New York, NY, USA, 2006. ACM Press.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD '79*, pages 23–34, 1979.

[29] Dave D. Straube and M. Tamer. Query optimization and execution plan generation in object-oriented data management systems. *IEEE*, pages 210–227, 1995.

[30] R. Tibshirani T. Hastie and J.H. Friedman. *The Elements of Statitical Learning.* Springer, New York, 1st edition, 2001.

[31] TPC. *TPC-H Benchmark.* Transaction Processing Performance Council, http://www.tpc.org/tpch, 2005.

[32] Robert Vieira. *Begining SQL Server 2005 Programming.* Wrox, 2006.

[33] Kyu-Young Whang and Ravi Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Trans. Database Syst.*, pages 67–95, 1990.

[34] Wikipedia. *WIKIPEDIA la enciclopedia libre.* http://es.wikipedia.org, 2007.