TÉCNICAS PARA LA MEJORA DEL JOIN PARALELO Y DEL PROCESAMIENTO DE SECUENCIAS TEMPORALES DE DATOS

JOSEP AGUILAR SABORIT

DAMA UPC - www.dama.upc.edu
Departamento de Arquitectura de Computadores
Universidad Politécnica de Cataluña
Barcelona (España), mayo , 2006
Director: Josep.L Larriba-Pey

"All endings are also beginnings. We just do not know it at the time", (Mitch Albom, *The five people you meet in heaven*)

Dedico esta Tesis doctoral a mi padre, a mi madre, y a mi hermano. Sin su apoyo, ayuda, y paciencia, nunca lo hubiera logrado.

Os quiere, Josep.

AGRADECIMIENTOS

Gracias en especial a mi director de Tesis Josep.L Larriba Pey, Larri. Él me abrió un camino lleno de oportunidades y siempre se comportó como un amigo. Muchas gracias Larri.

Luego también agradecer a Victor Muntés Mulero su ayuda y amistad a lo largo de estos 10 años: nos conocimos un primer dia del primer año de carrera, y juntos hemos finalizado el doctorado siempre en contínua colaboración y con muchas risas de por medio.

Agradecer, sin duda, el apoyo económico y logístico de IBM a través del proyecto *Center for Advanced Studies* (CAS IBM). Gracias a Calisto Zuzarte, Kelly Lyons, Wing Law, Miro Flasza, Mario Godinez, y Hebert Pereyra por su inestimable ayuda. También dar gracias al *Performance Team* y al *RunTime Team* de DB2, en especial a Berni Schiefer, Kelly Ryan, Haider Rizvi, Kwai Wong, y Adriana Zubiri.

ÍNDICE

AGRADECIMIENTOS v

LISTA DE TABLAS xi

LISTA DE FIGURAS xiii

1 INTRODUCCIÓN 1
- 1.1 Motivación 1
- 1.2 Objetivos 2
- 1.3 Contribuciones. Estructura de la Tesis 3

2 CONCEPTOS PRELIMINARES 5
- 2.1 Paralelismo en DataWarehouses y OLAP. Arquitecturas cluster 5
  - 2.1.1 Organización de los datos. Esquemas de tipo estrella 6
  - 2.1.2 Particionado de datos 7
  - 2.1.3 La operación de join paralela 8
- 2.2 Ejecución del join estrella 11
  - 2.2.1 El bitmap join (BJ) 11
  - 2.2.2 Multi Hierarchical Clustering (MHC) 13
- 2.3 Bit Filters 14
- 2.4 El algoritmo Hybrid Hash Join (HHJ) 15
  - 2.4.1 Uso básico de los bit filters en el algoritmo HHJ 17
  - 2.4.2 El algoritmo HHJ paralelo 17

3 REMOTE BIT FILTERS 19
- 3.1 Introducción 19
  - 3.1.1 Trabajo relacionado y motivación 19
  - 3.1.2 Contribuciones 20
- 3.2 Remote Bit Filters 20
  - 3.2.1 Remote Bit Filters Broadcasted (RBF_B) 21
  - 3.2.2 Remote Bit Filters with Requests (RBF_R) 21
- 3.3 Configuración de la evaluación 25
- 3.4 Resultados 27
- 3.5 Análisis comparativo 31
- 3.6 Conclusiones 35

4 PUSHED DOWN BIT FILTERS 37
4.1 Introducción 37
4.1.1 Trabajo relacionado y motivación 37
4.1.2 Contribuciones 38
4.2 Pushed Down Bit Filters (PDBF) 38
4.2.1 Contexto de ejecución 40
4.2.2 El algoritmo 41
4.3 Configuración de la evaluación 42
4.4 Resultados 44
4.5 Conclusiones 46
5 STAR HASH JOIN 51
5.1 Introducción 51
5.1.1 Trabajo relacionado y motivación 51
5.1.2 Contribuciones 52
5.2 Star Hash Join (SHJ) 52
5.2.1 Contexto de ejecución. Esquema de tipo Star Hash Join 53
5.2.2 Generalización de PDBF a arquitecturas cluster 54
5.2.3 El algoritmo 54
5.3 Configuración de la evaluación 56
5.4 Resultados 58
5.5 Conclusiones 64
6 MEMORY CONSCIOUS HYBRID HASH JOIN 69
6.1 Introdución 69
6.1.1 Trabajo relacionado y motivación 69
6.1.2 Contribuciones 70
6.2 Algoritmos paralelos del HHJ para SMP 71
6.3 El algoritmo Memory-Conscious 72
6.3.1 El algoritmo 72
6.3.2 Propiedades autonómicas 73
6.4 Configuración de la evaluación 74
6.5 Resultados 75
6.6 Conclusiones 80
7 SECUENCIAS TEMPORALES DE DATOS. LOS DYNAMIC COUNT FILTERS 83
7.1 Introducción 83
7.1.1 Trabajo relacionado y motivación 83
7.1.2 Contribuciones 85
7.2 Count Bloom Filters (CBF) 85
7.3 Spectral Bloom Filters 86

7.4 Dynamic Count Filters (DCF) 87
7.4.1 La estructura de datos 87
7.4.2 Consultar un elemento 88
7.4.3 Actualizaciones 89
7.4.4 Control de la operación de rebuild 90
7.5 DCF versión particionada (PDCF) 94
7.5.1 La estructura de datos 94
7.5.2 Actualización, consulta, y operación rebuild 96
7.5.3 Número óptimo de particiones (γ) 97
7.6 Comparativa entre SBF,DCF y PDCF 100
7.6.1 Recursos de memoria 100
7.6.2 Tiempo de acceso para localizar un contador 102
7.6.3 Operaciones de actualización 102
7.6.4 Rebuilds de la estructura 102
7.7 Configuración de la evaluación 103
7.8 Resultados 104
7.9 Conclusiones 111
8 CONCLUSIONES 113
REFERENCIAS 117
A MODELO ANALÍTICO PARA LOS REMOTE BIT FILTERS 123
B MODELO ANALÍTICO PARA EL STAR HASH JOIN 127
C MODELO ANALÍTICO PARA EL HYBRID HASH JOIN 131
D ARTÍCULOS PUBLICADOS DURANTE EL DESARROLLO DE ESTA TESIS 133

LISTA DE TABLAS

4.1 Configuraciones realizadas. 44

5.1 Tabla de grupos de selectividades aplicadas a SHJ. 57

5.2 Tabla de los atributos que aparecen en la consulta para nuestro análisis, y por los que la técnica MHC se ha dimensionado. 58

5.3 Características de cada técnica. 64

7.1 Comparación cualitativa de CBF, SBF, DCF y PDCF. 84

7.2 Resultados para PDCF. 108

LISTA DE FIGURAS

1.1	Estructura de la Tesis.	3
2.1	Ejemplo de sistema *cluster* con 3 nodos con configuración SMP.	6
2.2	Base de datos TPC-H y ejemplo de consulta SQL de join estrella. SF es el factor de escala aplicado a la base de datos.	7
2.3	TPC-H. Esquema de particionado Hash.	9
2.4	Join no colocalizado entre las tablas $lineitem$ y $supplier$.	10
2.5	Esquema estrella usando índices *bitmap*.	11
2.6	Ejecución del *bitmap join*.	12
2.7	*Multi Hierarchical Clustering* (MHC).	13
3.1	Contexto de ejecución.	22
3.2	Idea básica.	22
3.3	Esqueleto final de RBF_R.	23
3.4	Plan de ejecución.	26
3.5	Ejecución secuencial. Porcentajes de reducción de RBF_R y RBF_B respecto la ejecución *baseline*. (a) Porcentaje de reducción en la carga de la capa de comunicación; (b) Porcentaje de reduccion en el tiempo de ejecución.	28
3.6	Reducción del tiempo de ejecución de RBF_R y RBF_B respecto a la ejecución *baseline* cuando ejecutamos varias consultas de forma concurrente. Las lineas de barras son los resultados de las ejecuciones reales. Las lineas sólidas son los resultados obtenidos por el modelo analítico. $*$ significa que las consultas no pudieron ser ejecutadas debido a falta de memoria.	30
3.7	Reducción de carga en la capa de comunicación de RBF_R y RBF_B respecto la ejecución *baseline*. $*$ significa que las consultas no pudieron ser ejecutadas debido a falta de memoria.	31
3.8	Diferencia entre RBF_B y RBF_R en términos del porcentaje ahorrado en comunicación de datos. x es la relación entre el tamaño de la tupla y el tamaño de un código de hash.	33
3.9	Diferencia entre RBF_B y RBF_R en términos de utilización de los recursos de memoria. n es el número de valores distintos provenientes de la relacion de build.	34
4.1	Ejemplo de la técnica PDBF.	39
4.2	Aplicabilidad de los PDBF.	40
4.3	Consulta utilizada y su plan de ejecución.	43

4.4 Tiempo de ejecución y tráfico de datos entre nodos para las configuraciones #1, #2, #3 y #4. 47

4.5 Tiempo de ejecución y tráfico de datos entre nodos ara las configuraciones #5, #6 y #7. 48

4.6 E/S para los nodos hash join. 49

5.1 Esquema *Star Hash Join* para un *cluster* con k nodos. 53

5.2 (a) *Remote Bit Filters Broadcasted* (RBF$_B$); (b) *Pushed Down Bit Filters* (PDBF). 54

5.3 El algoritmo Star Hash Join. 55

5.4 Plan de ejecución. 57

5.5 (a) Reducción de datos comunicados entre nodos; (b) Reducción de E/S realizada por los operadores de join. 60

5.6 Ahorro de E/S en el *scan* sobre la tabla de hechos. 61

5.7 (a) Comunicación de datos entre nodos; (b) E/S realizada. 62

5.8 Escalabilidad de la E/S. 63

5.9 (a) Memoria utilizada por la técnica *Star Hash Join*; (b) Análisis de los bit filters. E/S realizada. 65

5.10 Solución híbrida. (a) Comunicación de datos; (b) E/S realizada. 67

6.1 Implementaciones *Basic* y NMC del algoritmo HHJ. Asumimos 4 procesos y i=1..B. 72

6.2 El algoritmo *Memory-Conscious*. 74

6.3 (a) Número de *bucket overflows*; (b) Tiempo de ejecución de la fase de join del algoritmo HHJ. 76

6.4 Resultados para la fase de join cuando ocurren *bucket overflows*. 77

6.5 Contención de memoria. 77

6.6 Tiempo de ejecución de la tercera fase del HHJ, la fase de join. 78

6.7 Escalabilidad. Tiempos de ejecución. 79

6.8 Tiempo de ejecución de la tercera fase del HHJ, la fase de join. 79

6.9 (a) Costes de E/S y cpu de cada algoritmo bajo los efectos de sesgo en los datos; (b) Contención de memoria. 81

7.1 Estructuras de datos de los SBF. 87

7.2 La estructura de datos DCF. 88

7.3 Contador de niveles. 91

7.4 Número de operaciones de *rebuild* del DCF para los tres distintos escenarios: T1 ($\frac{n_{inserts}}{n_{deletes}} > 1$), T2 ($\frac{n_{inserts}}{n_{deletes}} \simeq 1$), and T3 ($\frac{n_{inserts}}{n_{deletes}} < 1$). 92

7.5 Número de operaciones de *rebuild* en el DCF para diferentes valores de λ, y para diferentes *ratios* que siguen una distribución Zipfian con θ entre (a) 0-2 y (b) 1-2. R es el *ratio* $\frac{n_{inserts}}{n_{deletes}}$ utilizado para la la ejecución dada. 93

7.6 *Partitioned Dynamic Count Filters* (PDCF). La idea conceptual. 94

7.7 PDCF. La estructura de datos. 96

7.8 Visión de los PDCF. 96

7.9 (a)Tiempo de ejecución de la operación de *rebuild* ; (b) Memoria utilizada. Resultados para distribuciones uniformes y distribuciones no uniformes. 98
7.10 Uso de la memoria (a) SBF contra DCF; (b) SBF contra PDCF; (c) DCF contra PDCF. 101
7.11 Tiempo de ejecución medio para una operación de lectura, escritura, y de *rebuild*. 105
7.12 (a) Tiempo de ejecución; (b) Uso de la memoria. 106
7.13 Resultados del análisis dinámico. 107
7.14 Calidad de servicio. Tiempo requerido para responder consultas a lo largo del tiempo de ejecución para cada una de las 3 estrategias evaluadas. 109
7.15 (a)Uso de la memoria; (b) calidad de representación (CR_PDCF = CR_DCF). 110
8.1 Objetivos y técnicas. 113

1

INTRODUCCIÓN

1.1 Motivación

Un Sistema Gestor de Bases de Datos (SGBD) es la herramienta que gestiona la información almacenada en una base de datos, ya sea para realizar consultas, actualizaciones o modificaciones. Los SGBDs se han convertido en una importante fuente de investigación y desarrollo encaminada a encontrar nuevas técnicas que hagan de los mismos unas herramientas lo más eficientes posible.

Aplicaciones involucradas en grandes volúmenes de datos como *DataWarehousing* o *On-line Analytical Processing* (OLAP), a menudo requieren de la ayuda de hardware para un procesado eficiente de los datos. En muchos casos, grandes empresas utilizan arquitecturas paralelas para una eficiente ejecución de consultas complejas, que tienen fines análiticos dentro de sus negocios sobre grandes cantidades de datos.

El buen rendimiento de la ejecución de consultas en este tipo de entornos es clave por razones obvias: la decisión de un analista de negocios puede ser errónea si la respuesta obtenida durante la consulta no es actual. Ciertos aspectos son de vital importancia cuando se intenta obtener un buen rendimiento de una base de datos: la arquitecutra del computador, el diseño del SGBD y las virtudes del administrador de la base de datos.

Desde el punto de vista del arquitecto de redes y computadores, mejorar el rendimiento del hardware y software pueden ayudar. Sin embargo, aunque el hardware puede estar diseñado para explotar al máximo sus recursos, la importancia de diseñar un software consciente de la arquitectura hardware sobre la cual se ejecuta, puede ser decisiva en algunos casos, y dar pie a una importante mejora del rendimiento del SGBD permitiendo la ejecución paralela de un mayor número de consultas.

Una gran parte de esta Tesis se centra en la mejora de la operación de join paralela siendo conscientes del hardware sobre el cual se está ejecutando el SGBD paralelo. Dentro del álgebra relacional, la operación más compleja y costosa de llevar a cabo por un SGBD, es la operación de join. Los métodos más conocidos y extendidos en la literatura para una ejecución rápida y eficaz de esta operación, son: *Merge Sort Join*, *Nested Loop Join* y *Hash Join* [44]. Las dos primeras, para una ejecución eficiente, requieren de un cierto orden en los datos de sus relaciones fuente. Esto implica una ordenación previa o bien el soporte de una estructura auxiliar que de por si mantenga el orden de dichos datos. Por el contrario, la operación *Hash Join* no precisa de ningún orden sobre las relaciones fuente y utiliza algoritmos basados en hash para resolver el join. Varios

estudios [17; 36; 50] demuestran que los algoritmos basados en hash obtienen un mejor rendimiento, siendo la operación de Hash Join una de las más utilizadas por los SGBDs modernos. En esta Tesis, aunque generalizable a otros algoritmos de join, basa las técnicas propuestas en el algoritmo *Hybrid Hash Join* [36] paralelo. Proponemos técnicas para aliviar la capa de comunicación cuando éste se ejecuta sobre arquitecturas distribuidas, o bien reducir el procesamiento masivo de datos y mejorar la jerarquía de memoria cuando se ejecuta sobre arquitecturas paralelas con recursos compartidos.

Aparte de proponer técnicas para mejorar el rendimiento del join paralelo, esta Tesis también pone empeño en otro tema puntal dentro del tratamiento de datos, y de alta importancia en areas financieras y de telecomunicaciones, el procesado de secuencias temporales de datos. En dichos escenarios los datos evolucionan rápida y dinámincamente a lo largo del tiempo de ejecución de una aplicación. Los valores de los datos insertados y borrados a lo largo del tiempo de vida de una aplicación necesitan ser monitorizados, ya sea para tener un recuento aproximado del número de apariciones de un valor, o para saber si un valor pertenece o no al conjunto de datos representado. La monitorización rápida y eficiente de datos en entornos red, o cualquier otro entorno que requiera gestionar un flujo intenso de datos, se convierte en un requisito de gran importancia [39; 41; 56].

Para este tipo de situaciones se requiere de estructuras de datos especiales que aporten una solución computacionalmente rápida y baja en consumo de memoria. En [40] se proponen los *Count Bloom Filters* (CBF), que ofrecen una solución estática y carece de adaptabilidad y de precisión en la información representada. Una representación alternativa a los CBF, son los *Spectral Bloom Filters* [27] (SBF), que en este caso sí que proponen una solución dinámica, pero los métodos de acceso son lentos y la estructura es difícil de mantener. Nosotros presentamos los *Dynamic Count Filters* [7], que consisten en una nueva representación de los CBF que, a la vez de adquirir la adaptabilidad de los SBF, son rápidos en los accesos de lectura y escritura, y son fáciles de mantener.

1.2 Objetivos

Dentro del mundo del procesamiento de datos, la Tesis tiene múltiples objetivos sobre dos areas claramente diferenciadas, las bases de datos, y el procesado de secuencias temporales de datos.

Bases de datos. Se quiere estudiar y analizar diversas técnicas que tienen como finalidad mejorar el rendimiento de la operación de join paralela en los SGBDs modernos. Las técnicas objeto de estudio tendrán como objetivos :

- Reducir la comunicación de datos en arquitecturas hardware paralelas sin recursos compartidos. La transmisión de datos entre los nodos en arquitecturas distribuidas se convierte en un cuello de botella cuando una operación no se pueden ejecutar localmente en un nodo, y necesita realizar un uso intensivo de la red de interconexión. Así pues, la descongestión de la capa de comunicación se traduce en una mejora global en el rendimiento del SGBD.
- Mejorar la jerarquía de memoria en configuraciones hardware paralelas con recursos compartidos. Con ello se pretende reducir las operaciones de E/S, el consumo de cpu y la sincronización y contención de memoria entre procesos.
- Reducción del volumen de datos intra-operacional del plan de ejecución durante la ejecución de una consulta con múltiples operaciones de join. Esto se traduce en un menor coste por parte de los operadores del plan de ejecución y, en consecuencia, en

una disminución de la carga del SGBD que puede destinar más recursos a operaciones paralelas que se estén ejecutando en ese momento.

Secuencias temporales de datos. En el campo del procesamiento de secuencias temporales de datos se quiere conseguir una representación eficiente, en espacio y tiempo, de un conjunto de datos que evoluciona dinámicamente en el tiempo. Las operaciones de inserción y borrado son abundantes y muy frecuentes en el procesamiento de secuencias temporales de datos. Así pues, con el fin de obtener una representación precisa de dicho conjunto de datos, se requiere de estructuras adaptables, eficientes en espacio, y con rápidos métodos de acceso.

1.3 Contribuciones. Estructura de la Tesis

En el anterior apartado hemos mencionado los objetivos de la Tesis. A continuación definiremos cuales son las técnicas que nos disponemos a estudiar, con que fin, y cómo se relacionan entre ellas.

En el segundo Capítulo de la Tesis daremos un repaso a los conceptos preliminares y necesarios para una mejor comprensión lectora. Definiremos que es un *Datawarehouse* [25; 24] y que son las aplicaciones de tipo *Online Analytical Processing* [28] (OLAP), así como la estrecha relación entre estas aplicaciones y el uso de arquitecturas paralelas. También introduciremos la operación de join estrella, frecuente en los entornos *Datawarehouse* y OLAP, y uno de los principales focos de investigación en los últimos años. Las técnicas presentadas en esta Tesis para mejorar la ejecución del join paralelo, se centran en el uso de los *Bloom filters* [14], también conocidos como *bit filters*, y la implementación de la operación de join mediante el algoritmo *Hybrid Hash Join* [36]. Así pues, también daremos una explicación detallada de ambos conceptos.

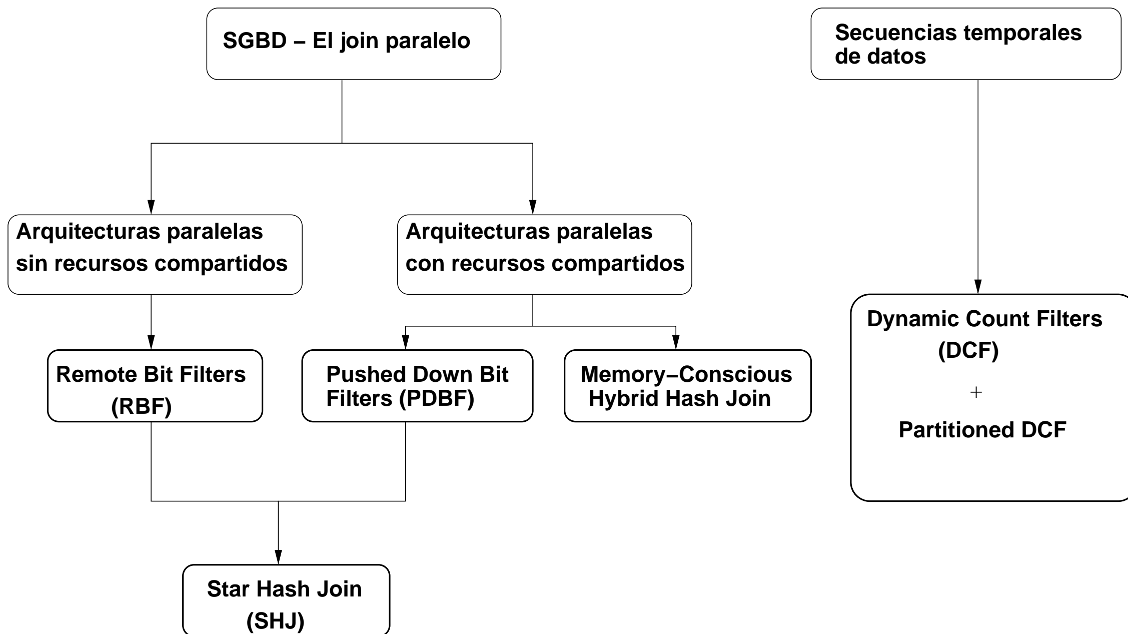


Figura 1.1 Estructura de la Tesis.

En los cinco Capítulos siguientes se explican todas y cada una de las técnicas expuestas en esta Tesis. La Figura 1.1 da una estructura de las cinco técnicas que presentamos, y como se relacionan entre sí.

Dentro del marco de las bases de datos y el rendimiento de la operación de join paralela en SGBDs modernos, presentamos 4 técnicas distintas. Orientado a arquitecturas hardware paralelas sin recursos compartidos, es objeto de estudio la técnica *Remote Bit Filters* [5; 6] (RBF). Esta técnica propone un método que, mediante el uso de los *bit filters* y un uso eficiente de la memoria, reduce la comunicación de datos entre nodos durante la ejecución de operaciones de join paralelas en entornos distribuidos. Para sistemas paralelos, y también secuenciales, con recursos compartidos, se proponen los *Pushed Down Bit Filters* [4] (PDBF), que aprovechando los *bit filters* creados en los operadores de join, tiene como objetivo reducir el volumen de datos a procesar por las operaciones del plan de ejecución de una consulta.

Los RBF y los PDBF, se unen para dar lugar al *Star Hash Join* [9] (SHJ). El SHJ es una técnica destinada a acelerar la ejecución del join estrella en arquitecturas paralalelas distribuidas, en la que los nodos de los que se compone tienen una configuración hardware paralela con recursos compartidos. En nuestro trabajo en particular, contemplamos una configuración *cluster* donde cada nodo tiene una arquitectura hardware con multiprocesadores simétricos (SMP). En una arquitectura paralela de este tipo, parte del potencial reside en sacar el máximo rendimiento de cada uno los nodos que la componen. Por ello, presentamos un algoritmo paralelo para optimizar la operación *Hybrid Hash Join* en nodos con configuraciones paralelas con recursos compartidos, el algoritmo *Memory-Conscious Hybrid Hash Join* [8].

Dentro del marco de secuencias temporales de datos exponemos un extenso Capítulo que extiende el uso de los *bit filters* a este ámbito y presenta la propuesta de los *Dynamic Count Filters* (DCF) [7], y de su variante particionada, los (Partitioned DCF). Los DCF son una representación de los *Count Bloom Filters* [40], similares a los *bit filters*, pero substituyendo cada bit, por un contador de bits. Los DCF, siendo eficientes en espacio de memoria y tiempo de ejecución, quieren dar una representación fiable de un conjunto de datos que evoluciona dinámicamente a lo largo del tiempo. Los PDCF que se presentan como una versión optimizada de los DCF, aportan mejoras tanto en espacio como en tiempo de ejecución respecto a los DCF, dando en definitiva una mejor calidad de servicio.

En los Apéndices A B, y C, se proponen modelos analíticos para los *Remote Bit Filters*, la ejecución paralela de *Hybrid Hash Join*, y la ejecución del join estrella respectivamente. Los modelos analíticos elaborados son una importante cotribución de esta Tesis, el hecho de que se coloquen en los Apéndices es meramente una decisión de estructura del documento presentado.

Los dos primeros modelos se validan en sus respectivos Capítulos donde se explican las técnicas asociadas. Los modelos ayudan a (i) reafirmar las conclusiones obtenidas a partir de los resultados experimentales, y (ii) simular la ejecución de dichas técnicas bajo situaciones extremas. En el caso del modelo para la ejecución del join estrella, éste nos ayuda a analizar la técnica SHJ bajo un entorno de ejecución concreto, y a realizar un profundo análisis comparando la técnica SHJ con otras dos técnicas destinadas a mejorar la ejecución del join estrella, el *bitmap join* [64], y el *Multi Hierarchical Clustering* [57].

Cada Capítulo en sí introduce las conclusiones que hemos obtenido de cada técnica presentada y analizada. Igualmente, como Capítulo final a este documento de Tesis, daremos las conclusiones finales de todo el trabajo realizado a lo largo de estos casi 5 años.

2

CONCEPTOS PRELIMINARES

2.1 Paralelismo en DataWarehouses y OLAP. Arquitecturas cluster

Un *DataWarehouse* [25; 24] es una base de datos que colecta y almacena datos provenientes de múltiples fuentes, remotas, y heterogéneas de información. La información de un *DataWarehouse* no es volátil, y la cantidad de datos almacenados se acerca a los PetaByte de forma constante. Los *DataWarehouses* dan soporte a tecnologías OLAP [28] (*On-Line Analytical Processing*), para que de forma eficiente, permitan a los analistas, managers, y ejecutivos, extraer la información necesaria para la toma de decisiones en entornos empresariales. El procesamiento de consultas de tipo OLAP son de un coste computacional muy elevado, y con un acceso masivo a disco debido al gran flujo de datos procesado. Este tipo de entornos requiere de poderosas arquitecturas paralelas con el fin de obtener un eficiente rendimiento global del sistema.

Multiprocesadores simétricos con memoria compartida (SMP), han sido ampliamente utilizados para mejorar el throughput en sistemas de bases de datos paralelos [23; 38]. En un sistema SMP, todos los procesadores comparten recursos de disco y memoria bajo el control de una copia del sistema operativo. Los procesadores acceden a la memoria a través de buses de alta velocidad y modernas redes de interconexión. Sin embargo, estas arquitecturas presentan problemas de escalabilidad debido a limitaciones de disco, de memoria o de contención en la red.

Cuando los *DataWarehouses* han de escalar por encima del número de procesadores que puede proporcionar una arquitectura SMP, o cuando las aportaciones de un sistema de altas prestaciones son deseables, entonces, las arquitecturas *cluster* son la opción escogida [35]. El uso de arquitecturas *cluster* se ha convertido en una solución muy común para el soporte de aplicaciones que requieren de paralelismo masivo y, en entornos *DataWarehouse*, se ha hecho imprescindible para alcanzar un buen rendimiento. En noviembre del 2004, cerca del 60% de los supercomputadores en la lista 'TOP500' [1] eran arquitecturas *cluster*, alcanzando un 72% en el 2005. Este tipo de arquitecturas se basan en un diseño hardware sin recursos compartidos [79], y están compuestas de varios nodos que se comunican entre ellos a través de una red de interconexión. En dichos sistemas, la base de datos está particionada horizontalmente [3] entre los nodos del sistema. El particionado horizontal permite que, durante la ejecución de una consulta, se pueda realizar una distribución equitativa del trabajo. De esta forma forma, se alcanza un óptimo rendimiento cuando cada nodo puede trabajar de forma local con su partición de la base de datos, y con la mínima comunicación de datos posible.

Los nodos con una configuración SMP se presentan como los bloques óptimos para construir un sistema *cluster* [71] (véase Figura 2.1). Las arquitecturas formadas por varios nodos SMP, también conocidas como arquitecturas CLUMP [29], ofrecen una gran escalabilidad y efectividad, siendo comúnmente utilizadas en entornos *DataWarehouse*. Los resultados de este tipo de arquitecturas en sistemas OLAP bajo el *benchmark* de bases de datos TPC-H pueden consultarse en [2].

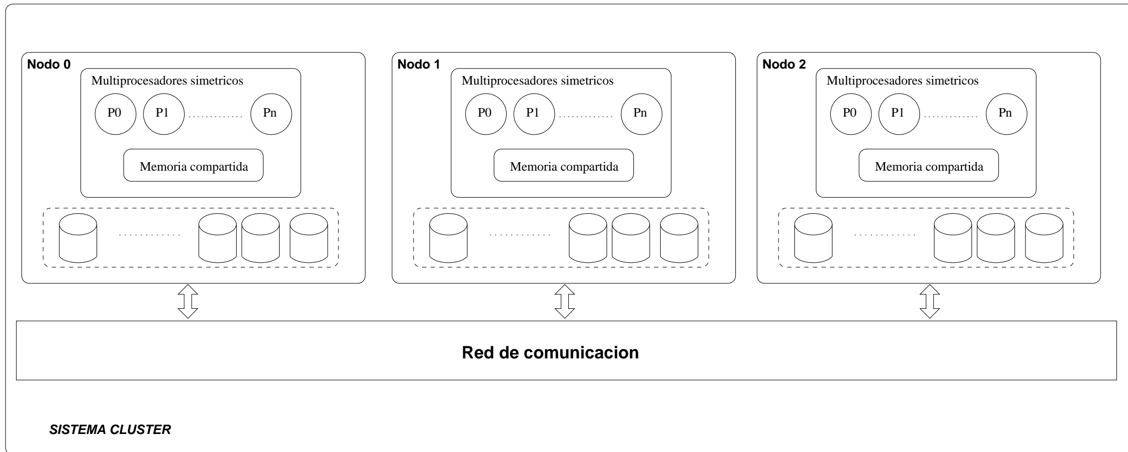


Figura 2.1 Ejemplo de sistema *cluster* con 3 nodos con configuración SMP.

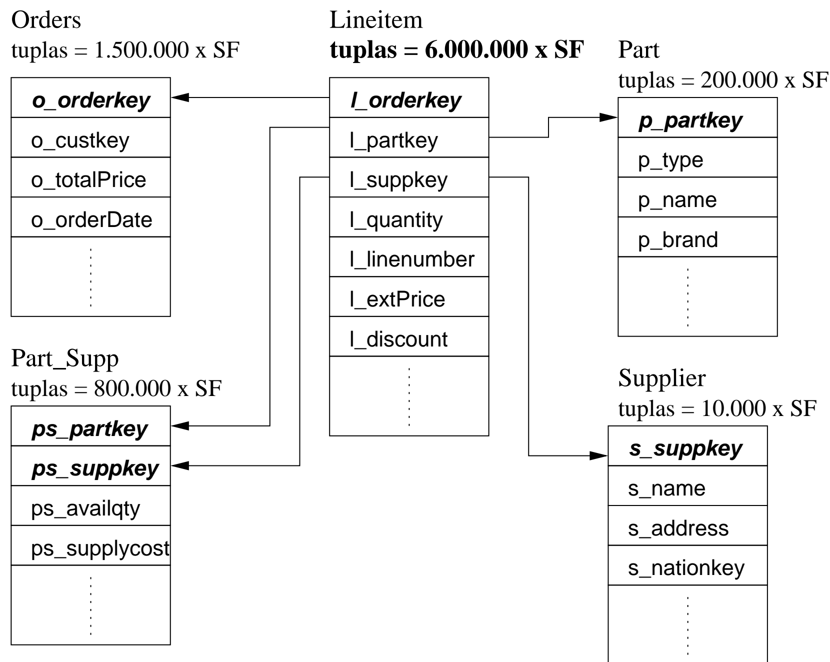
2.1.1 Organización de los datos. Esquemas de tipo estrella

Normalmente, los entornos *DataWarehouse* están organizados acorde a un modelo multidimensional consistente en uno o varios esquemas de tipo estrella [24]. Cada esquema estrella consiste en una tabla central (tabla de hechos) rodeada por múltiples tablas que la dimensionan (tablas de dimensión). La tabla de hechos está conectada a las tablas de dimensión mediante relaciones de clave primaria y clave foránea. La Figura 2.2 ilustra un ejemplo de esquema estrella basado en la base de datos de TPC-H [2].

TPC-H modeliza el análisis en la fase final del entorno de los negocios, en el que se intenta dar soporte a la toma de decisiones robustas. La Figura 2.2 muestra las principales tablas que forman la base de datos de TPC-H. La tabla *lineitem* es la tabla de hechos y mantiene la información de cada linea de compra procesada. La tabla de hechos conecta las cuatro tablas de dimensión *supplier*, *part*, *part_supp*, y *orders*, que dan información de las compras, productos, proveedores y pares producto/proveedor respectivamente.

Las consultas de tipo OLAP suelen ser complejas y ad hoc con factores de selectividad muy altos, véase la consulta SQL mostrada en la Figura 2.2 a modo de ejemplo. Este tipo de consultas no son conocidas a priori y suelen ejecutar múltiples join que relacionan la tabla de hechos con sus respectivas tablas de dimensión. Este tipo de consultas se denominan consultas ad hoc de tipo join estrella [25]. En este tipo de consultas, diferentes predicados son aplicados sobre las tablas de dimensión, acotando el resultado de la consulta que suele ser agrupado con fines analíticos. La inmensa cantidad de datos a extraer de la tabla de hechos se convierte en el principal cuello de botella durante la ejecución de este tipo de consultas, y su rapidez de procesado pasa a ser crucial en entornos *DataWarehouse*.

TPC–H . Esquema multidimensional




```
select
p_name, p_partkey, sum(l_quantity)
from
    part,
    lineitem,
    partsupp,
    orders
where
    ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
    and o_orderdate between date '1996-01-01' and date '1997-12-31'
    and ps_availqty> '10'
    and p_name like 'azure'
group by
    p_name, p_partkey
```

Figura 2.2 Base de datos TPC-H y ejemplo de consulta SQL de join estrella. SF es el factor de escala aplicado a la base de datos.

2.1.2 Particionado de datos

Originariamente [75], el particionado de una relación implica distribuir sus tuplas a través de los múltiples discos en una máquina sin recursos compartidos. De esta forma, el particionado horizontal de los datos [3] permite a las bases de datos paralelas explotar el ancho de banda de la

E/S, leyendo y escribiendo en paralelo sobre múltiples discos. La misma filosofía se extiende a las arquitecturas *cluster* pero substituyendo discos por nodos.

El particionado de una base de datos en arquitecturas *cluster* puede seguir diferentes esquemas, que son los encargados de decidir el nodo destino de cada una de las tuplas almacenadas en las tablas de la base de datos.

Los esquemas de particionado más utilizados son:

- **Particionado Round-Robin**. Particiona de forma equitativa las tuplas de la base de datos entre los *clusters* del sistema sin tener en cuenta la naturaleza, ni los valores, de los datos almacenados. *Round-Robin* permite un balanceo de carga equitativo para accesos secuenciales a las tablas de la base de datos.

- **Particionado por rango.** Particiona las tablas de la base de datos según los rangos establecidos sobre los valores de uno de sus atributos. El principal problema de este tipo de particionado es que no es equitativo, y puede provocar un claro desbalanceo de carga entre los nodos del sistema.

- **Particionado hash.** Particiona las tablas de la base de datos a través de una función de hash que se aplica sobre los valores de uno o varios atributos de cada tabla. El atributo/s sobre los que se aplica la función de hash se denominan *clave de particionado*. Por cada tupla de una tabla, la función de hash retorna un valor que especifica el nodo destino en el que se ha de almacenar. De esta forma se logra una distribución equitativa, y con conocimiento del valor de los datos por los que se particiona la base de datos.

La Figura 2.3 ilustra el particionado hash del esquema estrella de la Figura 2.2 sobre una arquitectura *cluster* con 4 nodos. Las claves de particionado sobre las que se aplica el particionado son *o_orderkey*, *l_orderkey*, *p_partkey*, *ps_partkey/ps_suppkey* y *s_suppkey* para las tuplas de las tablas *orders, lineitem, part, partsupp* y *supplier* respectivamente. La función de hash retorna un valor entre 0 y 3 que determina el nodo en el que se debe de almacenar cada tupla. Si la función de hash es lo suficientemente precisa y asumiendo una distribución uniforme de los datos, entonces se obtiene un buen balanceo de los datos entre los nodos del sistema.

2.1.3 La operación de join paralela

Cuando se ejecuta una operación de join sobre arquitecturas *cluster*, y asumiendo un esquema de particionado hash como el descrito anteriormente, entonces, se dice que el join es *colocalizado* si la clave de join es la misma que la clave de particionado de las tablas involucradas en la operación. De la misma forma, se dice que una tabla está colocalizada si su clave de particionado coincide con la clave de la operación de join. Cuando el join es colocalizado, se puede realizar la ejecución de forma local en cada nodo, pues, en este caso, el particionado hash nos asegura que las tuplas de un nodo no harán join con las tuplas de un nodo remoto. Si la clave de particionado no coincide con la clave de join, entonces, se dice que el join es *no colocalizado*, y precisa de comunicación de datos para su ejecución. El cómo se comunican los datos durante la ejecución de un join no colocalizado depende del estado de las tablas involucradas en la operación:

- **Una de las dos tablas no está colocalizada**. En este caso, bien se reparticiona la tabla no colocalizada por la clave de join, o bien se realiza un *broadcast* de una de las dos tablas. La decisión entre una opción u otra la toma el optimizador en función de su coste. Notar que la

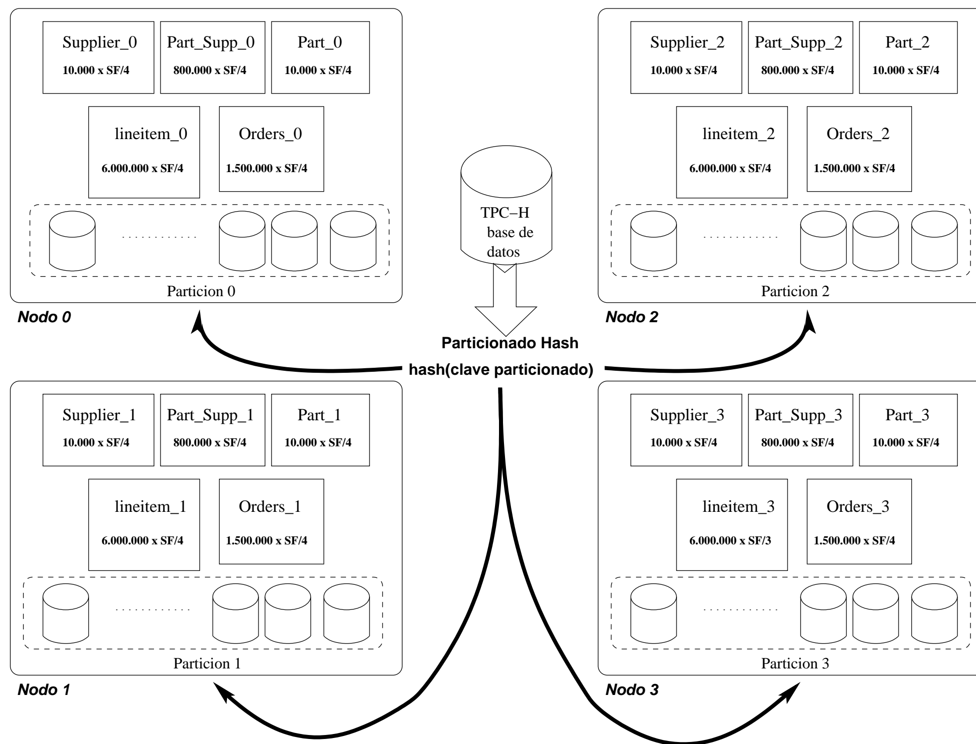


Figura 2.3 TPC-H. Esquema de particionado Hash.

operación de *broadcast* suele ser de un coste muy elevado, y en este caso, la mejor opción suele ser reparticionar la tabla no colocalizada.

- **Las dos tablas no están colocalizadas**. Para este caso, bien se reparticionan las dos tablas por la clave de join, o bien se realiza el *broadcast* de una de las dos tablas. De nuevo la decisión la toma el optimizador en función del coste de cada una. En este caso, sin embargo, puede ser que el reparticionado de dos tablas resulte más costoso que realizar la operación *broadcast*.

En general, la paralelización de un join cuando las relaciones están colocalizadas es simple: el plan de ejecución se replica en todos los nodos, de forma que hay un operador de join en cada nodo ejecutándose en paralelo sobre la partición de la base de datos que le corresponde. La paralelización de un join no colocalizado añade más complejidad al plan de ejecución. En este caso, un nuevo operador de reparticionado es añadido para comunicar los datos de las relaciones entrantes del join no colocalizado.

Cada operador de reparticionado debe de llevar a cabo dos acciones diferentes: enviar datos al resto de los nodos del sistema, así como recibir datos de cada uno de estos nodos. En el primero de los casos, el operador de reparticionado actúa como el *sending end*, y en el segundo caso actúa como el *receiving end*. El operador de reparticionado tiene conocimiento del esquema de particionado llevado a cabo y de la topología de los nodos involucrados en la ejecución de la consulta. De esta forma, para cada tupla proyectada por los nodos inferiores del plan de ejecución local, el operador de reparticionado decide si la tupla tiene que ser procesada de forma local (es proyectada

directamente al operador local immediatamente superior del plan de ejecución), o si tiene que ser procesada por uno o más nodos (es enviada a través de la red de interconexión).

A modo de ejemplo, y retomando el esquema de tipo estrella y el correspondiente particionado mostrados en las Figuras 2.2 y 2.3, sólo el join entre las relaciones $orders$ y $lineitem$ ($o_orderkey = l_orderkey$) sería colocalizado puesto que las claves de particionado son $o_orderkey$ y $l_orderkey$ respectivamente. El resto de joins entre la tabla de hechos y las tablas de dimensión son no colocalizados. Esta es una situación común en un esquema de tipo estrella, donde las claves de particionado son las claves primarias para las tablas de dimensión, y la clave foránea de la tabla de dimensión más grande para la tabla de hechos. De esta forma, se intenta minimizar la comunicación de datos durante la ejecución de un join estrella. La Figura 2.4 muestra el join no colocalizado entre las relaciones $supplier$ y $lineitem$ ($l_suppkey = s_suppkey$) de la base de datos TPC-H. La tabla $lineitem$ no está colocalizada, y es la que se reparticiona de forma selectiva (punto a punto) por el atributo $l_suppkey$ a través del operador de reparticionado RO.

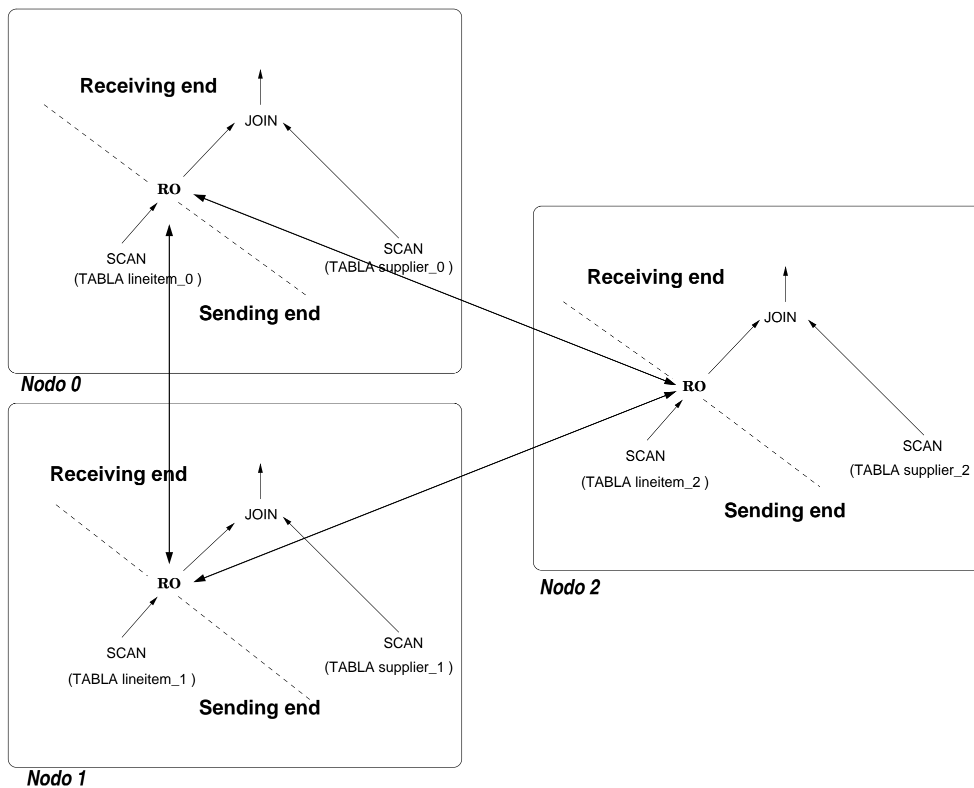


Figura 2.4 Join no colocalizado entre las tablas $lineitem$ y $supplier$.

Notar que si el SGBD lo soporta, y si la capacidad de almacenamiento, la sobrecarga en la administración, y la sobrecarga en la sincronización, no son de factores a tener en consideración, entonces las tablas de dimensión pueden ser replicadas en su completitud en cada nodo antes de que las consultas sean ejecutadas. En este caso, la comunicación de datos en consultas de tipo join estrella no es necesaria. En nuestro trabajo, se asume que las tablas de dimensión no están replicadas en cada nodo por falta de recursos.

2.2 Ejecución del join estrella

A continuación explicamos dos técnicas destinadas a acelerar la ejecución de consultas de la operación join estrella. Primero explicamos el *bitmap join* [64], que lo introducimos como la base teórica para la ejecución óptima de un join estrella. El *bitmap join* reduce al mínimo posible los datos a procesar de la tabla de hechos, y basa su ejecución en operaciones entre mapas de bits en memoria. Sin embargo, su ejecución literal en SGBDs reales no es factible debido a la gran cantidad de recursos necesitados por la técnica. Así pues, se proponen otras técnicas como el *Multi Hierarchical Clustering* [57], que a través de un camino distinto y más práctico, trata de implementar la misma idea que el *bitmap join*.

2.2.1 El bitmap join (BJ)

El *bitmap join* (BJ) introducido por O'Neil y Graefe en [64], utiliza índices *bitmap* [22; 83] para ejecutar el join estrella a través de rápidas operaciones de bits, que resultan en un vector de bits que indica cuales son las tuplas de la tabla de hechos que forman parte del resultado final. La idea conceptual de BJ es, reducir la cantidad de datos a procesar de la tabla de hechos a las tuplas que satisfacen la consulta.

En su forma más simple, un índice *bitmap* sobre una tabla T aplicado sobre uno de sus atributos, consiste en una lista de tuplas por cada valor del atributo representado. Cada lista se representa a través de un *bitmap* o vector de bits, con un total de $\|T\|$ posiciones, donde $\|T\|$ es la cardinalidad de la tabla T. Cada posición del vector de bits toma como valor 1 si la tupla asociada está contenida en la lista representada, de otra forma toma valor 0. Normalmente, el *row identifier*, comúnmente conocido como RID, es el método utilizado para mapear cada posición del vector de bits y las tuplas indexadas. Usando esta idea, el join entre dos tablas T y S, sobre un atributo en común, puede representarse a través de un índice *bitmap* (llamado índice *bitmap join*) con un tamaño de $\|T\| \times \|S\|$ bits. Por cada tupla de T, necesitamos un vector de $\|S\|$ bits indicando con que tuplas de S hace join.

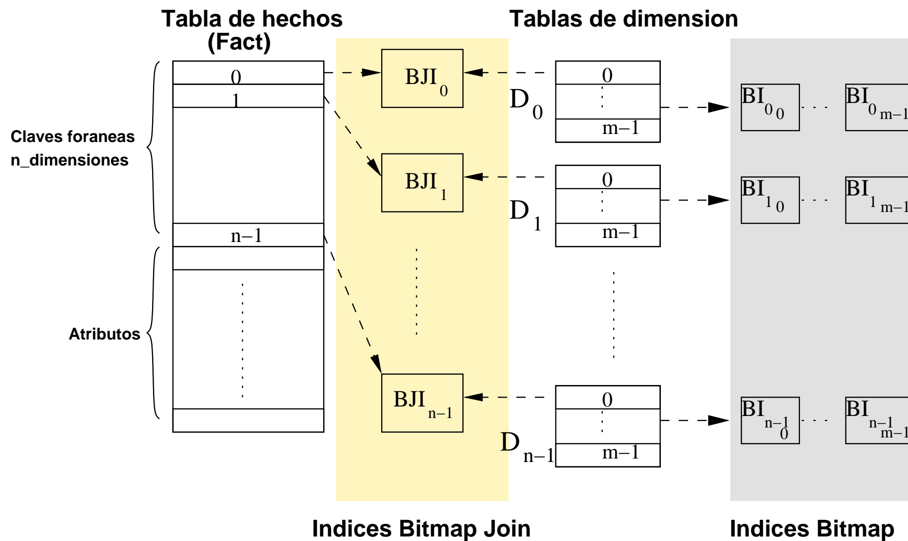


Figura 2.5 Esquema estrella usando índices *bitmap*.

Usando este tipo de índices podemos tener un esquema estrella organizado tal y como se muestra en la Figura 2.5 :

- una tabla de hechos ($Fact$) y n tablas de dimensión $D_i : i = 0..n-1$. Cada tabla de dimensión tiene $c = 0..m-1$ columnas, donde m puede ser diferente para cada tabla.
- un índice *bitmap* por cada tabla de dimensión D_i y cada columna c, $BI_{i_c} : i = 0..n-1 \wedge c = 0..m-1$.
- un índice *bitmap join* para cada join entre la tabla de hechos y las tablas de dimensión D_i, $BJI_i : i = 0..n-1$. Todos los join entre D_i y $Fact$ son indexados y tienen un tamaño de $\|D_i\| \times \|Fact\|$ cada uno.

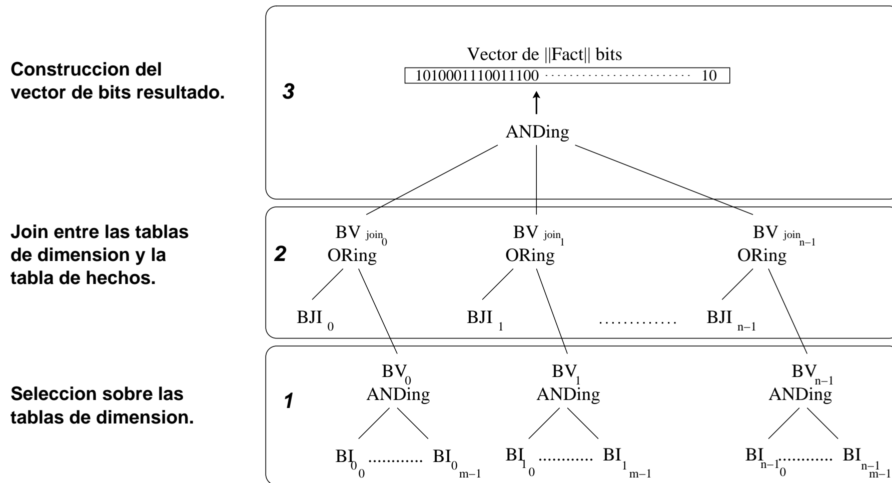


Figura 2.6 Ejecución del *bitmap join*.

Con este esquema estrella, una consulta de tipo join estrella entre n tablas de dimensión y la tabla de hechos, puede ser ejecutado a través de rápidas operaciones de bits realizadas en memoria. El mecanismo de ejecución se muestra en la Figura 2.6, y se puede explicar como sigue:

1. Selección sobre las tablas de dimensión. Asumiendo que cada tabla de dimensión D_i tiene una o más restricciones en cada columna c, para $c = 0..m-1$, la selección de las tuplas de una tabla de dimensión D_i se realiza a través de la intersección de los índices *bitmap* $\bigcap_{c=0}^{m-1} BI_{i_c}$. El resultado para cada tabla de dimensión, es un vector de bits BV_i de tamaño $\|D_i\|$ indicando las tuplas de D_i que cualifican.
2. Cada bit en el vector BV_i representa una tupla de la tabla de dimensión D_i, y tiene asociada una lista de las tuplas de la tabla de hechos con las que hace join en el índice *bitmap join* BJI_i. Recordar que cada lista está representada como un vector de bits de $\|Fact\|$ posiciones. Entonces, para esos bits de BV_i que tienen como valor 1, se aplica ORing sobre los vectores de bits asociados en BJI_i. Para cada join entre una tabla de dimensión D_i y la tabla de hechos $Fact$, el resultado es un vector de bits BV_{join_i} de tamaño $\|Fact\|$ bits que indica las tuplas de la tabla de hechos que hacen join con las tuplas selecionadas de D_i.

3. Finalmente se aplica ANDing $\bigcap_{i=0}^{n-1} BV_{join_i}$, y el resultado es un vector de bits de $\|Fact\|$ posiciones, que nos indica las tuplas de la tabla de hechos que forman parte del resultado de la consulta.

Después de los pasos indicados arriba, BJ obtiene un un vector de bits indexado por RID, donde cada bit mapea una tupla de la tabla de hechos. Si el valor del bit es 1, entonces la tupla mapeada ha de ser procesada, de otra forma, la tupla se descarta. El join estrella se completa realizando el join de las tuplas de la tabla de hechos que satisfacen la consulta, con las tablas de dimensión involucradas en el resultado final. Para acelerar este último paso se pueden utiliar índices join [65], de forma que tanto el *scan* sobre la tabla de hechos, como el join sobre las tablas de dimensión no son necesarios.

2.2.2 Multi Hierarchical Clustering (MHC)

Las arquitecturas *cluster* descomponen grandes problemas en pequeños fragmentos, de modo que, cada fragmento del problema puede ser ejecutado en paralelo por cada nodo. Para el caso específico de la ejecución de un join estrella, la tabla de hechos se particiona entre los nodos del sistema, de modo que el procesado del gran volumen de datos a ser extraído de la tabla de hechos es compartido por todos los nodos. Además de este particionado horizontal entre nodos, aparecen nuevas estrategias de particionado físico internas a cada nodo [31; 57; 62].

La partición de la tabla de hechos en cada nodo se puede realizar especificando varios atributos. De esta forma, la tabla de hechos está organizada respecto múltiples atributos jerárquicos que la dimensionan. A esta nueva organización de la tabla de hechos se la denomina *Multi Hierarchichal Clustering* [57] (MHC). La Figura 2.7 muestra un ejemplo de MHC para el esquema estrella propuesto en la Figura 2.2 basado en la base de datos TPC-H. En este ejemplo, para cualquier nodo, la tabla de hechos *lineitem* está organizada a través de tres dimensiones jerárquicas: *o_orderdate*, *p_brand* y *s_nationkey*.

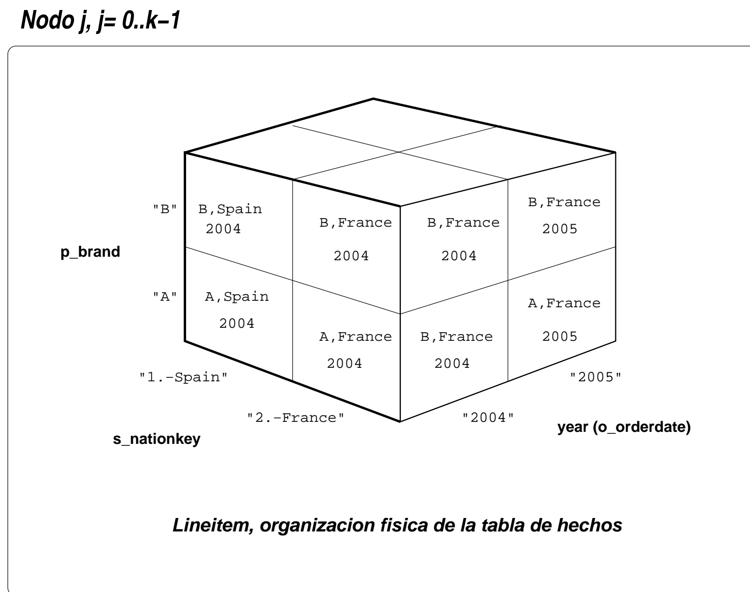


Figura 2.7 *Multi Hierarchical Clustering* (MHC).

Los beneficios del *clustering* jerárquico para consultas de tipo join estrella fué observado por primera vez en [31], donde el espacio multidimensional de la consula se divide de forma uniforme entre varias particiones de una granularidad menor que el nivel de memoria cache para la consulta dada. En [57] se utilizan claves surrogate para codificar de forma eficiente las dimensiones jerárquicas, y los índices multidimensionales UB-Tree [11], que permiten un acceso punto a punto para datos multidimensionales, se utilizan como organización primaria de la tabla de hechos. De esta forma, los join de tipo estrella se convierten en consultas por rango sobre múltiples dimensiones, reduciendo así la E/S sobre la tabla de hechos. En [62] se propone un sistema de ficheros específico para OLAP basado en *chunks*. En este caso, la consulta de tipo join estrella es, de nuevo, convertida en una consulta por rangos en el espacio de datos multidimensional y multinivel de un cubo, sobre el cual, el acceso a los datos es proveído por el sistema de almacenamiento de los mismos.

En [48] se presenta un completo y detallado plan de ejecución para el procesado de consultas ad hoc de tipo join estrella sobre tablas de hechos multidimensionadas jerárquicamente. Bajo un esquema denormalizado, se codifican de forma jerárquica las tablas de dimensión aplicando *surrogates*, de modo que cada clave surrogate para cada tabla de dimensión se almacena en la tabla de hechos. Cuando se procesa una consulta de tipo join estrella, se crean rangos sobre cada clave surrogate de las tablas de dimensión que estan restingidas en la consulta. Estos rangos definen uno o más hiper-rectángulos en el espacio multidimensional de la tabla de hechos, de modo que el número de accesos sobre la tabla de hechos se reduce de forma muy significativa.

En los trabajos previos mencionados [31; 57; 62] se proponen distintos esquemas para acceder de forma eficiente a la tabla de hechos codificada de forma jerárquica. El SGBD producto de IBM, DB2 UDB tiene su propio esquema de acceso a los datos jerárquicos, y consiste en una organización orientada a bloques sobre las tablas multidimensionadas de forma jerárquica [66; 67]. Una única combinación de los valores de las tablas de dimensión está físicamente organizada como un bloque de páginas, siendo un bloque de un grupo de páginas consecutivas en disco. Se crean *block indices* para acceder a los bloques, y los métodos de acceso son extendidos con el fin de extraer los datos con gran rápidez.

Esquemas MHC son buenos para consultas de tipo join estrella cuando la selectividad de la consulta es relativamente pequeña. De este modo el coste de la consulta reside en leer la tabla de hechos. Sin embargo tienen el inconveniente de que los atributos utilizados para multidimensionar la tabla de hechos tienen que coincidir con los que restringen las tablas de dimensión en la consulta. Cuando nos referimos a consultas ad hoc, aumenta la probabilidad de que los atributos que acotan la consulta no coincidan con los de la técnica MHC.

2.3 Bit Filters

El *Bloom filter*, también conocido como *bit filter* o *hash filter*, fué inventado por Burton Bloom en 1970 [14], y básicamente consiste en un vector de bits de m posiciones que representa un conjunto de n mensajes, $S = s_1, ...s_n$. El *bit filter* utiliza d funciones de hash $h_1, h_2, ..., h_d$ que determinan d entradas del bit filter. Cada función de hash retorna un valor de 1 a m, de tal forma que para cada mensaje $s \in S$, las posiciones del *bit filter* $h_1(s), h_2(s), ..., h_d(s)$, inicialmente con valores igual a 0, toman como valor un 1. Diferentes mensajes pertenecientes a S pueden tener asignadas posiciones comunes en el bit filter, así pues, un mensaje pertenece a S, $s \in S$, con una probabilidad de error P si $\forall i : i = 1..d : h_i(s) = 1$. Es decir, si se da el caso en que $\exists i : i = 1..d : h_i(s) = 0$, entonces podemos asegurar que $s \notin S$. En caso contrario $s \in S$ con una probabilidad de falso positivo P.

Bloom teoriza sobre la probabilidad de falsos positivos dependiendo del tamaño del *bit filter* m, del número de valores distintos n, y del número de bits d utilizados en el *bit filter* para cada mensaje. Una vez todos los mensajes en S se han insertado aleatoriamente en el bit filter, entonces, la probabilidad p de que un bit en particular sea 0 es:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{dn}\right) \simeq e^{-dn/m} \tag{2.1}$$

Así pues, la probabilidad de error del *bit filter* es:

$$P = (1 - p)^d \tag{2.2}$$

Los análisis presentados en [61; 73] demuestran que el número de funciones de hash d que minimiza la probabilidad de falsos positivos viene dado por:

$$k = (\ln 2) \times \frac{m}{n} \tag{2.3}$$

A la práctica, es preferible un valor pequeño para d ya que supone menor coste computacional.

Posteriormente a la contribución de Bloom, se ha realizado mucha investigación relativa a los *bit filters* en diferentes áreas de la computación tanto hardware [78] como software [40].

Dentro del área de gestión y procesado de información, el uso de los *bit filters* ha sido especialmente útil para acelerar la ejecución de la operación de join [45; 77] y la ejecución de consultas con múltiples operaciones de join [4; 26; 49]. También, durante la ejecución de la operación de join en sistemas paralelos distribuidos, los *bit filters* se han utilizado para evitar comunicación de datos y reducir el tiempo de computación [33; 34; 52; 55].

2.4 El algoritmo Hybrid Hash Join (HHJ)

En entornos secuenciales, estudios como [17; 36; 50] demuestran que los algoritmos de join basados en hash, no sólo son fáciles de paralelizar, sino que además, de entre todos ellos, el algoritmo *Hybrid Hash Join* [36] (HHJ) es el que mejor rendimiento obtiene.

El algoritmo HHJ fué descrito por primera vez en [36]. HHJ es un algoritmo basado en hash, variante del algoritmo *GRACE Hash Join* [43; 50]. A lo largo de esta Sección denominamos R y S a las relaciones sobre las que se va a realizar la operación de join. Asumimos que los datos de R y S tienen una distribución uniforme. Denotamos por $|R|$ y $|S|$ el tamaño, en páginas de memoria, que ocupan las relaciones R y S respectivamente, y por $|M|$ la cantidad de páginas de memoria disponibles por el algoritmo. Asumimos que R es más pequeña que S, $|R| < |S|$.

El algoritmo HHJ sobre las relaciones R y S puede ser descompuesto en tres fases:

1. **Fase de build**. En esta fase se particiona la relación R en un grupo de $B + 1$ *buckets* disjuntos: $R_0, R_1, ..., R_B$. La primera fase debe de asegurar que cada *bucket* cabe en la memoria asignada $|M|$ para la operación. Definimos un factor de fuga F para contabilizar el espacio ocupado por la tabla de hash y otras estructuras adicionales necesarias para gestionar un *bucket* $|R_i|$. Así pues, tal y como se especifica en [36], el número de *buckets* a crear se calcula como:

$$B = max\left(0, \frac{|R| \times F - |M|}{|M| - 1}\right) \tag{2.4}$$

El particionado se realiza aplicando una función de hash sobre la clave de join de cada tupla perteneciente a R con lo que las tuplas con idéntico atributo de join serán almacenadas en el mismo *bucket*. Durante la ejecución de esta fase, las tuplas del primer *bucket* se utilizan para construir una tabla de hash en memoria, mientras que los B *buckets* restantes son almacenados en disco. La única restricción durante el desarrollo de esta fase es la cantidad mínima de memoria necesaria para gestionar los B *buckets*:

$$|M| \geq |R_0| \times F + B \tag{2.5}$$

donde almenos B páginas de memoria son necesarias para mantener un *buffer* de salida para cada *bucket*.

2. **Fase de probe**. En esta fase se particiona la relación S utilizando la misma función de hash y rangos de partición que en la fase de *build*. De este modo, sólo es necesario realizar el join sobre las tuplas locales a cada par de *buckets* $\langle R_i, S_i \rangle$ para $i = 0..B$. El join de las tuplas pertenecientes a S_0 se realiza de forma inmediata ya que en la fase de *build* se construyó la tabla de hash para el *bucket* R_0. Los B *buckets* restantes son, de nuevo, almacenados en disco.

3. **Fase de join**. En esta fase se realiza el join de los B *buckets* de la relación R y S almacenados en disco. El join de cada par de *buckets* $\langle R_i, S_i \rangle$ se realiza en dos fases:

   (a) Se construye una tabla de hash con las tuplas del *bucket* R_i.

   (b) Se leen las tuplas del *bucket* S_i y se chequean contra la tabla de hash en busca de tuplas que hagan join.

   Cuando el HHJ no se puede ejecutar dentro de la memoria asignada $|M|$ (ejecución *out of core*), los *buckets* pueden ser relativamente grandes en número y tamaño, de modo que en estos casos, la fase de join del algoritmo puede ser considerada como la de un coste más elevado.

Problema: *Bucket Overflow*

La fase de particionado debe de asegurar que cada *bucket* R_i quepa en la memoria compartida por todos los procesos participantes en la operación de join. Sin embargo, esto no es siempre posible debido a la presencia de sesgo en los datos, errores en los cálculos del tamaño de las particiones, funciones de hash poco precisas, o escasez de memoria. Así pues, nos podemos encontrar con el caso de que algunos *buckets* crezcan de forma desmedida, y no haya memoria suficiente para cargarlos durante la fase de join del algoritmo. Denominamos a este suceso como *bucket overflow*. Los *bucket overflows* son solventados utilizando el algoritmo *hashed loops* [17], que es una variación del algoritmo *Nested Loop Join*. El algoritmo *hashed loops* realiza $\left\lceil \frac{|R_i|}{|M|} \right\rceil$ iteraciones, de tal forma que cada iteración realiza el join de un grupo de tuplas de R_i con todas las tuplas de S_i, siguiendo los dos pasos de la fase de join del HHJ. El algoritmo *hashed loops* tiene un coste muy elevado tanto de E/S como de tiempo de cpu. Esto hace que minimizar el número de *bucket overflows* se convierta en un punto crucial para obterner un buen rendimiento del algoritmo HHJ.

2.4.1 Uso básico de los bit filters en el algoritmo HHJ

Los *bit filters* son originariamente utilizados durante la ejecución del algoritmo HHJ para ahorrar cómputo de datos y E/S [77]. El algoritmo HHJ crea el *bit filter* previo inicio de la fase de *build*. Si aislamos m de la ecuación 2.2 obtenemos:

$$m = \frac{1}{\left(1 - ((1 - P^{1/d})^{1/dn})\right)} \tag{2.6}$$

A través de esta ecuación se observa claramente que el *bit filter* se dimensiona según (i) el número de valores distintos n del atributo de join proyectados de la relación R, (ii) el número de funciones de hash d utilizadas, y (iii) la fracción de falsos positivos P establecida, en este caso, por el optimizador. Cada posición del *bit filter* es inicializada a cero.

El uso del *bit filter* se limita sólo a las dos primeras fases del algoritmo HHJ:

- durante la fase de *build* se aplican d funciones de hash sobre el atributo de join de cada tupla proyectada de la relación R. Denominamos al resultado de aplicar una función de hash sobre la clave de join, *código de hash*. Cada código de hash determina una entrada en el *bit filter* que cambiar su valor a uno.

- durante la fase de *probe* se aplican las mismas d funciones de hash sobre el atributo de join de cada tupla proyectada de la relación S. Sólo que una de las d entradas del *bit filter* esté a cero, entonces, sabemos con total certeza que esa tupla no va a realizar join con ninguna tupla de la relación R y que, por lo tanto, se puede descartar de immediato y ahorrarnos su procesado. Si las d entradas del *bit filter* están a uno, entonces se procede al procesado de la tupla pues hará join con una probabilidad de error P.

Selectividad de un bit filter

Definimos la selectividad de un *bit filter* S_{bf} como la fracción de tuplas de la relación de *probe* que no son filtradas. Asumiendo que los valores de la relación de *build* son únicos, entonces, S_{bf} está directamente relacionado con la fracción de falsos positivos P y la selectividad de la relación de *build* S_R a partir de la cual se creó el bit filter. Así pues, calculamos la selectividad de un *bit filter* como:

$$S_{bf} = S_R + (1 - S_R)P \tag{2.7}$$

El valor perfecto para S_{bf} sería aquel que coincidiera con S_R, es decir, aquel en el que $P = 0$. Sin embargo, debido a la presencia de falsos positivos, tenemos que contabilizar la fracción de tuplas de la relación de *probe* que pasan el *bit filter* y no hacen join con ninguna tupla de la relación de *build*: $(1 - S_R)P$.

2.4.2 El algoritmo HHJ paralelo

Arquitecturas cluster

La paralelización del algoritmo HHJ en arquitecturas *cluster* sigue las pautas explicadas en la Sección 2.1.3. Si la operación HHJ es no colocalizada, entonces se realiza la repartición de los datos no colocalizados, o se procede a realizar el *broadcast* de una de las dos relaciones. Como se ha mencionado anteriormente en este Capítulo, la operación de *broadcast* suele tener un coste

excesivo, y a no ser que los datos a comunicar sean pocos, la decisión más común a tomar por el optimizador es realizar el reparticionado de los datos de la relación no colocalizada (ya sea la relación de *build*, o la relación de *probe*). En este caso, la creación de los *bit filters* durante la fase de *build* se realiza sólo sobre la parte de la relación almacenada en el nodo de forma local. Por lo tanto, el uso de los *bit filters* para el algoritmo HHJ en arquitecturas *cluster* es el mismo que el explicado en la Sección anterior para entornos secuenciales, y se limita sólo a los datos que han de ser procesados por el nodo que los ha creado.

Arquitecturas SMP

Para explicar el algoritmo HHJ en arquiteturas SMP, nos remitimos al trabajo realizado en [54] en el que se realiza un modelo analítico de los algoritmos *GRACE Hash Join*, *Hybrid Hash Join* y *hashed loops join* sobre arquitecturas SMP. En el caso del algoritmo HHJ, todos los procesos participantes trabajan juntos durante toda la ejecución del join. Durante las fases de *build* y *probe*, todos los procesos trabajan en paralelo para particionar las relaciones R y S, y durante la fase de join, todos los procesos ejecutan un par de *buckets* $\langle R_i, S_i \rangle$ a la vez. De esta forma, se construye una tabla de hash global para cada *bucket* proveyendo mecanismos de lock para escrituras en la tabla de hash y, permitiendo que puedan darse varias lecturas en paralelo. Los resultados para esta propuesta muestran que, al contrario de lo que ocurría para entornos secuenciales, el algoritmo HHJ no obtiene siempre el mejor rendimiento. Cuando el número de *buckets* es pequeño, la contención de memoria añadida en las dos primeras fases hacen que el algoritmo empeore el rendimiento. Para solventar este promblema se propone una versión modificada del *Hybrid Hash Join*: para las dos primeras fases de *build* y *probe*, se utilizan *p buffers* de salida para cada *bucket*, donde *p* es el número de procesos que ejecutan el algoritmo. De esta forma se elimina la contención para las dos primeras fases del algoritmo durante el particionado de R y S. Esta versión modificada del HHJ siempre rinde mejor excepto en el caso en que las relaciones R y S tengan un tamaño similar, en este caso el algoritmo *hashed loops* obtendría un mejor rendimiento.

En [63] se estudia el problema de contención de memoria para el algoritmo *GRACE Hash Join* cuando hay sesgo en los datos. Aunque este trabajo se centra en el algoritmo *GRACE Hash Join*, podría ser aplicado a la fase de join del algoritmo HHJ. En esta propuesta los procesadores son balanceados entre los *buckets* dependiento del sesgo en los datos. Sabiendo el tamaño de cada *bucket* y dividiéndolo por el tamaño uniforme de un *bucket*, se decide el número de procesos asignado a cada *bucket*. A la par, las páginas de las relaciones R y S están estratégicamente particionadas entre los discos del sistema, de modo que, la contención se minimiza debido a que múltiples procesos leen distintas páginas en distintos discos al mismo tiempo.

3

REMOTE BIT FILTERS

3.1 Introducción

Aplicaciones que tratan con grandes volúmenes de datos como *DataWarehousing* o *On-line Analytical Processing* (OLAP) requieren, a menudo, de la ayuda de arquitecturas *cluster* para obterner una eficiente ejecución de consultas complejas, que suelen tener fines analíticos para la toma de decisiones en los negocios. Como se vió en el Capítulo 2, este tipo de arquitecturas consisten en un conjunto de nodos conectados entre sí a través de una red de interconexión, cada nodo mantiene una porción base de datos, de tal forma que, durante la ejecución de una consulta se alcanza un buen rendimiento cuando los nodos del sistema pueden trabajar en paralelo con la mínima comunicación de datos posible.

En este Capítulo estudiamos el uso remoto de los *bit filters* como mecanismo de reducción de comunicación de datos en arquitecturas *cluster*. En concreto, nos centramos en la reducción de comunicación de datos durante la ejecución de joins no colocalizados en este tipo de arquitecturas.

3.1.1 Trabajo relacionado y motivación

Los *bit filters* [14] han sido utilizados como mecanismo para el ahorro de comunicación de datos. Por ejemplo, lo operación de Semijoin [13; 12] es una de las estrategias más populares para el ahorro de comunicación en sistemas distribuidos. Supongamos dos relaciones R y S almacenadas en máquinas distintas cada una. El Semijoin de R a S se realiza proyectando R sobre los atributos de la clave de join, y de esta forma, la proyección resultante es enviada a la máquina donde S está almacenada, donde se determina las tuplas de S que hacen join con R. El resultado de este join, normalmente más pequeño que la relación S es enviado a la máquina donde reside R para completar la operación de join. La proyección de los atributos de R se representa mediante un vector de bits indexado por los valores de la clave de join [52]. Los *bit filters* [55] son utilizados como alternativa, y con el fin de reducir el tamaño de la proyección de los atributos de R. Esta estrategia, sin embargo, requiere que las tablas involucradas en el join estén en su completitud almacenadas en diferentes máquinas, además de la creación y el envío de los *bit filters* de una de las tablas en la operación de join.

En otras situaciones, se envía una copia de los *bit filters* a todos los componentes del sistema, como es el caso de la máquina de base de datos relacional GAMMA [33; 34]. Al contrario de lo que ocurre en sistemas distribuidos con la operación de Semijoin, la máquina GAMMA posee un

esquema centralizado en el que durante la operación *Hybrid Hash Join* [36], y una vez la fase de build ha finalizado, los *bit filters* son enviados a un proceso planificador que los envía, a su vez, a los procesos responsables de la ejecución de la fase de probe.

Además, los *bit filters* utilizados en estas técnicas pueden ser enviados de forma comprimida, tal y como se propone en [60], ahorrando de esta forma ancho de banda en la capa de comunicación.

El uso de los *bit filters* en arquitecturas *cluster* no ha sido propuesto hasta la fecha. Sin embargo, una extensión de las técnicas previamente explicadas a este tipo de arquitecturas consistiría en realizar una copia de los *bit filters* de cada nodo en todos los nodos del sistema. Llamamos a este uso de los *bit filters*, *Remote Bit Filters Broadcasted* (RBF$_B$).

RBF$_B$ tiene como principal desventaja el uso agresivo que realiza de la memoria al requerir que cada nodo almacene todos los *bit filters* del sistema, lo cual puede limitar el rendimiento del SGBD durante la ejecución de consultas complejas sobre grandes relaciones.

En este Capítulo proponemos un nuevo protocolo llamado *Remote Bit Filters with Requests*, RBF$_R$. El protocolo usa los *bit filters* en el contexto de joins no colocalizados. Por una parte, RBF$_R$ añade muy poca carga en la capa de comuncicación comparado con el uso normal de los *bit filters* especificado en la literatura [12; 33; 55]. Por otra parte, RBF$_R$ ahorra espacio en memoria puesto que evita el envío de los *bit filters* localmente creados en cada nodo por las operaciones de join involucradas en la consulta. En consecuencia se reduce la cantidad de memoria requerida para la ejecución de consultas, permitiendo un mejor uso de los recursos de memoria.

3.1.2 Contribuciones

Teniendo en cuenta el entorno de desarrollo al que nos referimos, podemos enumerar las principales contribuciones como:

1. La propuesta de *Remote Bit Filters with Requests* (RBF$_R$),un nuevo protocolo para el uso remoto de los bit filters en una arquitectura paralela distribuida.

2. La implementación de un prototipo de RBF$_R$ y RBF$_B$ en el Sistema Gestor de Bases de Datos IBM DB2 UDB v.8.1. Ninguno de ellos ha sido previamente implementado o analizado en un SGBD comercial.

3. La propuesta de tres modelos analíticos, para RBF$_R$, RBF$_B$ y el uso local de los *bit filters*, que validamos a lo largo del Capítulo. La comparación de los modelos da una idea cuantitativa de cual es la mejor estrategia en función de diferentes parámetros como las características de los datos, el SGBD y la arquitectura de la máquina utilizada.

4. El análisis del impacto de RBF$_R$ en una arquitectura *cluster* sobre una base de datos TPC-H de 100GB, y su comparación con la ejecución de otras estrategias.

5. Dos importantes conclusiones dentro del problema de procesado paralelo de consultas. Primero RBF$_R$ y RBF$_B$ reducen en igual proporción la carga sobre la capa de comuncicación, así como el coste de procesado de las consultas testeadas. Segundo, RBF$_R$ solventa los problemas de recursos de memoria contemplados en RBF$_B$ cuando se ejecutan múltiples consultas de forma concurrente.

3.2 Remote Bit Filters

Nos adentramos en el uso de *bit filters* durante la ejecución de joins no colocalizados en los que una, o las dos relaciones involucradas en la operación de join no han sido particionadas por la clave

de join. Tal y como se explicó en el Capítulo 2, cuando esto sucede, es necesario comunicar datos durante la ejecución del join.

Durante la paralelización de un join no colocalizado, la comunicación de datos tiene un gran impacto en el rendimiento de la operación de join. Minimizar la cantidad de datos a transmitir entre los nodos del sistema durante la ejecución del join es crucial para obtener un buen rendimiento.

Los *bit filters* en este caso pueden ser utilizados de forma local, de igual forma que en los joins colocalizados, filtrando tuplas al mismo tiempo que llegan al operador de join. En este caso, la relación reparticionada durante la operación de join se comunica en su totalidad. El uso local de los *bit filters* en joins no colocalizados es la implentación *baseline* en este Capítulo.

3.2.1 Remote Bit Filters Broadcasted (RBF$_B$)

Una posibilidad para evitar el total reparticionado de la relación de join es que cada nodo del sistema envíe una copia de sus *bit filters* al resto de nodos remotos [33; 34; 55]. Entonces, todos los operadores de reparticionado tienen acceso a todos los *bit filters* del sistema, de forma que cada tupla es chequeada contra el *bit filter* del nodo destino antes de ser enviada. Así pues, sólo aquellas tuplas que tienen posibilidad de hacer join son empaquetadas en un *buffer* de datos (uno por nodo destino) que será enviado cuando esté lleno de tuplas. Este método ahorra tiempo de comunicación, evitando el tráfico innecesario de datos a través de la red, y tiempo de proceso, evitando el empaquetado y desempaquetado de tuplas. Denominamos a esta estrategia *Remote Bit Filters Broadcasted* (RBF$_B$).

El principal problema de RBF$_B$ reside en que los *bit filters* pueden ser especialmente grandes en tamaño, lo cual implica un consumo elevado de los recursos de memoria por cada nodo al tener que mantener una copia de los *bit filters* del resto de los nodos del sistema. Asumiendo que tenemos un *heap* de memoria limitado para las operaciones de join, RBF$_B$ podría, fácilmente, quedarse sin memoria cuando varias operaciones de join se ejecutan de forma concurrente.

A continuación, proponemos *Remote Bit Filters with Requests* (RBF$_R$), un nuevo protocolo para el uso remoto de los *bit filters* evitando el problema de memoria mencionado anteriormente.

3.2.2 Remote Bit Filters with Requests (RBF$_R$)

La explicación detallada de la técnica usa como modelo de operación de join el *Hybrid Hash Join* (HHJ) paralelo. Sin embargo, nuestra propuesta podría ser aplicada sobre cualquier otra implementación de join por igualdad como el *Merge Sort Join* [36; 45].

Como se explica en el Capítulo 2, al ejecutar la operación HHJ no colocalizada, hay varias formas de decidir cómo las relaciones de build o probe deben de ser reparticionadas. En particular, explicamos el uso de RBF$_R$ cuando la relación de probe del algoritmo HHJ debe de ser reparticionada. Mostramos esta configuración en la Figura 3.1. En la Figura se observa la presencia de un operador de reparticionado (RO), que es el encargado de reparticionar los datos proyectados por la relación de probe, y por la relación de build en caso de que tampoco esté colocalizada (Figura 3.1-(b)).

Nótese que si se hiciera un *broadcast* de la relación de build a todos los nodos del sistema, entonces, no haría falta reparticionar la relación de probe ya que tendríamos la relación de build completa en cada nodo del sistema.

RBF$_R$

El protocolo que proponemos está esquematizado en la Figura 3.2. En un entorno paralelo, cada operador de reparticionado utilizará esta estrategia.

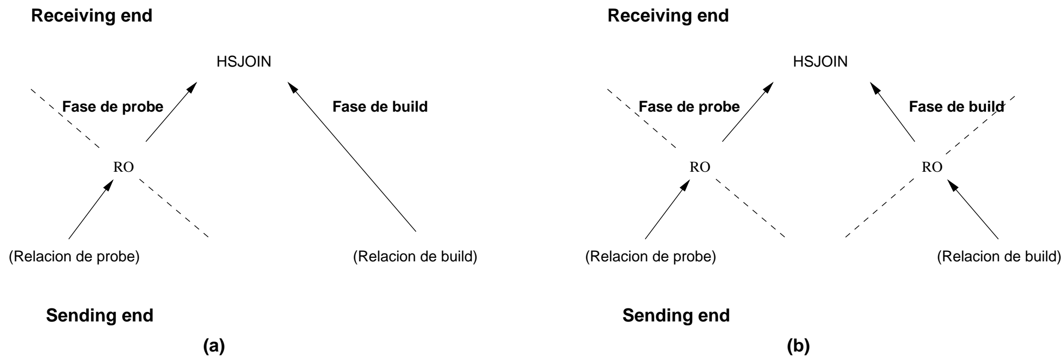


Figura 3.1 Contexto de ejecución.

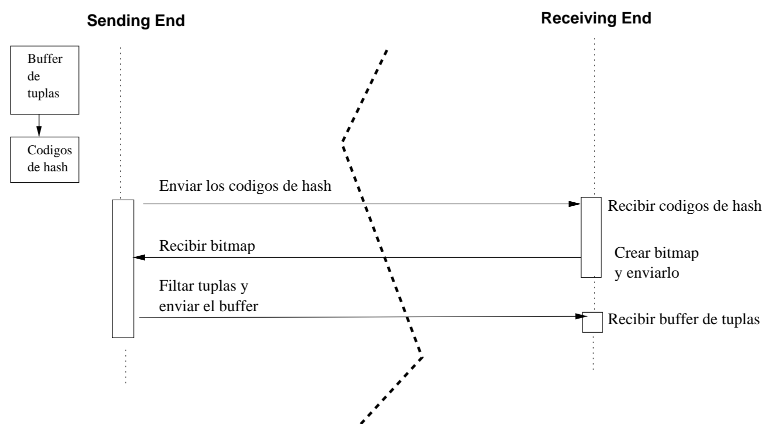


Figura 3.2 Idea básica.

Cuando una tupla que tiene que ser comunicada es procesada por el operador de reparticionado local, ésta se almacena en un *buffer* de datos asignado al operador de reparticionado del nodo destino. También, se crea el código de hash de la respectiva tupla y se almacena en un *buffer* separado (uno por cada posible nodo destino). Cuando el *buffer* de datos está lleno, el *buffer* que contiene los códigos de hash es enviado al operador de reparticionado del nodo destino. A partir de los códigos de hash recibidos, el nodo destino crea un *bitmap* basado en el resultado de chequear cada código de hash en el *bit filter* de su nodo local. Así pues, el *bitmap* consiste en un serie de bits, uno por cada código de hash, que indica si la tupla debe de ser enviada (bit con valor 1), o si debe de ser filtrada (bit con valor 0). El *bitmap* creado es enviado de vuelta al operador de reparticionado origen, que lo aplica sobre el *buffer* de datos y sólo aquellas tuplas que tienen posibilidad de hacer join son finalmente enviadas al operador de reparticionado del nodo destino.

Es importante resaltar que, normalmente, el tamaño de los *buffers* que contienen los códigos de hash, y los *buffers* que contienen los *bitmap*, son mucho mas pequeños que el tamaño de un *buffer* de tuplas. El tamaño de un código de hash es 4 bytes, y el tamaño de una entrada del *bitmap* es un bit.

Mejorando RBF$_R$

Proponemos dos mejoras con respecto la implementación de RBF$_R$ mostrada previamente:

1. Hacemos que todas las comunicaciones sean asíncronas, de modo que el envío de mensajes y el procesamiento de tuplas puede ser solapado.

2. Empaquetamos los *buffers* enviados entre los operadores de reparticionamiento en grupos de *buffers*. Es decir, sólo enviamos *buffers* con códigos de hash en el supuesto de que un total de GR *buffers* esté preparado para ser enviado a un cierto nodo destino. El número de *buffers* de datos enviados será menor que GR dependiendo de la selectividad de los *bit filters* S_{bf}.

Nuestro objetivo es reducir el núnero de *buffers* de códigos de hash y mensajes de *bitmaps* lo máximo posible, reduciendo así el número de mensajes de control necesarios para el envío de datos.

En la Figura 3.3 mostramos un ejemplo con el esqueleto final de la técnica. En el ejemplo utilizamos $GR = 4$ y los *bit filters* con $S_{bf} = 0.5$. En consecuencia, la mitad de las tuplas serán filtradas por el *sending end*, y sólo 1 de cada 2 *buffers* de datos serán enviados.

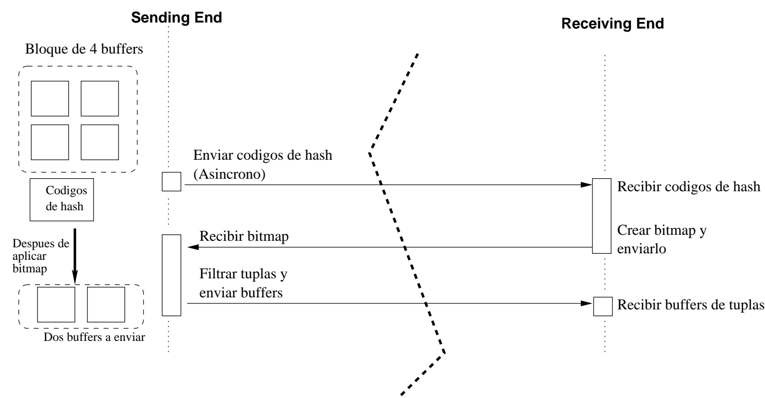


Figura 3.3 Esqueleto final de RBF$_R$.

Dimensionando el número de buffers para RBF$_R$

Los sistemas de comunciación permiten enviar *buffers* de datos de forma asíncrona, previniendo de esta forma el bloqueo de los *sending ends*. Sin embargo, al no haber una capacidad ilimitada para el almacenamiento de *buffers* en los *receiving ends* de una red de comunicación, entonces, debe de haber puntos de sincronización entre los *sending ends*. Denominamos a esta limitación de espacio "*capacidad del operador de reparticionado*", y denotamos el número de *buffers* de datos enviados entre dos puntos de sincronización por C. Derivado de la presencia de C, y de la selectividad de los *bit filters* S_{bf}, podemos limitar el número de *buffers* de datos enviados, de forma consecutiva, en un grupo de GR *buffers* para RBF$_R$ de la siguiente forma:

$$GR \leq \left\lfloor \frac{C}{S_{bf}} \right\rfloor \tag{3.1}$$

Asumimos que cada operador de reparticionado involucrado en RBF$_R$ tendrá un *heap* de memoria separado (S_{heap_R}) durante el desarrollo de la técnica. El tamaño del *heap* de memoria debe de ser fijado por el optimizador del SGBD. Así pues, si el S_{heap_R} disponible es lo suficientemente grande como para usar el máximo valor para GR definido en la ecuación 3.1, entonces obtenemos el valor óptimo para GR. Si esto no es posible, RBF$_R$ se adapta a la cantidad de memoria disponible fijando GR a un valor más pequeño.

Valor de GR

En esta Sección nos disponemos a formular el valor de GR en función del tamaño del *heap* (S_{heap_R}). Esto permitirá al optimizador escoger entre el óptimo GR, o el máximo GR acotado por el tamaño del *heap*.

Definimos las siguientes variables para un nodo:

n_{rro} número de operadores de reparticionado remotos.

$|B_d|$ número de tuplas que caben en un *buffer* de datos.

S_{hc} tamaño en bytes de un código de hash.

B_{d_s} tamaño en bytes de un *buffer* de datos.

Empezamos calculando el S_{heap_R} que necesita RBF$_R$. Necesitamos un grupo de *buffers* para cada una de las conexiones a las que estamos enviando datos, por esta razón, el tamaño del *heap* para un operador de reparticionado de RBF$_R$ depende del número de operadores de reparticionado remotos n_{rro}. También, necesitamos el doble de espacio para los *buffers* con el fin de mantener dos peticiones en proceso de forma concurrente. Debido a la naturaleza asíncrona de nuestra estrategia, empezamos llenando un nuevo grupo de *buffers* mientras la petición previa está en proceso, entonces, necesitamos mantener dos grupos de *buffers* por cada conexión: $2n_{rro}GRB_{d_s}$. Por cada petición en proceso necesitamos almacenamiento para el *buffer* de códigos de hash $|B_d|S_{hc}$ y para el *bitmap* que se tiene que recibir $\frac{|B_d|}{8}$.

$$S_{heap_R} = n_{rro} \times GR \times \left(2B_{d_s} + \frac{|B_d|}{8} + |B_d|S_{hc}\right) \tag{3.2}$$

Asumiendo que no tenemos espacio suficiente en el *heap* de memoria para utilizar el máximo valor para GR, podemos calcular GR a través de la ecuación 3.2:

$$GR = \left\lfloor \frac{S_{heap_R}}{n_{rro}\left(|B_d|S_{hc} + \frac{|B_d|}{8} + 2B_{d_s}\right)} \right\rfloor \tag{3.3}$$

Bit filters contra el tamaño de heap

Otro aspecto importante de cara a evaluar ambas técnicas, es comparar el *heap* de memoria necesitado por RBF_B (S_{heap_B}) con la memoria utilizada por RBF_R (S_{heap_R}) definida previamente.

RBF_B necesita almacenar los *bit filters* de los nodos remotos que intervienen en la operación del hash join. Siendo m el tamaño en bits de un bit filter, podemos definir el *heap* de memoria en bytes necesitado por RBF_B como :

$$S_{heap_B} = \frac{m}{8} n_{rro} \tag{3.4}$$

El tamaño de un *bit filter* m depende de el número de valores distintos n, la fracción de falsos positivos P, y el número de funciones de hash d. Asumiendo que P y d son valores fijados por el optimizador, entonces, a través de las ecuaciones 3.2 y 3.4 podemos observar que, mientras S_{heap_B} es proporcional a n y a n_{rro}, S_{heap_R} es proporcional a GR y a n_{rro}.

GR está limitado por la capacidad de la capa de comunicación C, y el número de valores distintos n está limitado por la cardinalidad de la relación de build. Así pues, es razonable asumir que $n \gg C$, concluyendo de esta forma que RBF_R siempre consume menos recursos de memoria que RBF_B.

A modo de ejemplo, teniendo la siguiente configuración para RBF_B y RBF_R: $n_{rro} = 7$, $P = \frac{1}{32}$, $d = 1$, $n = 10^7$, $GR = 5$, $|B_d| = 100$, $S_{hc} = 4$, y $B_{d_s} = 4KB$, obtenemos que para una sola consulta $S_{heap_B} = 262MB$, y $S_{heap_R} = 360KB$. En este caso, pues, RBF_R requiere del casi tres ordenes de magnitud menos memoria que RBF_B.

3.3 Configuración de la evaluación

Se ha implementado un prototipo de RBF_R y RBF_B sobre el Sistema Gestor de Bases de Datos DB2 UDB v8.1. Las ejecuciones se han realizado sobre una arquitectura formada por 8 nodos IBM p660s, conectados vía *ethernet* a un *switch* Cisco Blade de un Gigabit. Cada nodo IBM p660s tiene una configuración SMP con 4 procesadores Power_RS64-IV compartiendo 8GB de memoria. El sistema operativo de la plataforma es AIX 4.3.3.

Para la ejecución de nuestras pruebas la máquina se ha reservado de forma exclusiva, es decir, en tiempo de ejecución el sistema no está siendo compartido con ninguna otra aplicación.

Entorno de ejecución

Hemos utilizado la base de datos del *benchmark* TPC-H [2] con un tamaño de 100GB. Los datos han sido balanceados entre nodos utilizando un particionado de hash. La consulta SQL utilizada para nuestra evaluación se ha creado basada en las consultas propuestas por el mismo *benchmark*. La consulta que utilizamos sirve para mostrar el problema y entender las características de las estrategias evaluadas:

```
select distinct
    ps_partkey, ps_availqty, l_extendedprice,
    l_discount,l_shipmode
from
    tpcd.lineitem, tpcd.partsupp
where
```

```
        ps_partkey=l_partkey and
        l_suppkey=ps_suppkey
        and ps_availqty > 5000
order by
        ps_availqty, l_extendedprice
fetch first 10 rows only;
```

No discutiremos la semántica de la consulta pues no es de interés para evaluar nuestro trabajo. Más bien, nos centramos en el plan de ejecución generado por DB2 y mostrado en la Figura 3.4

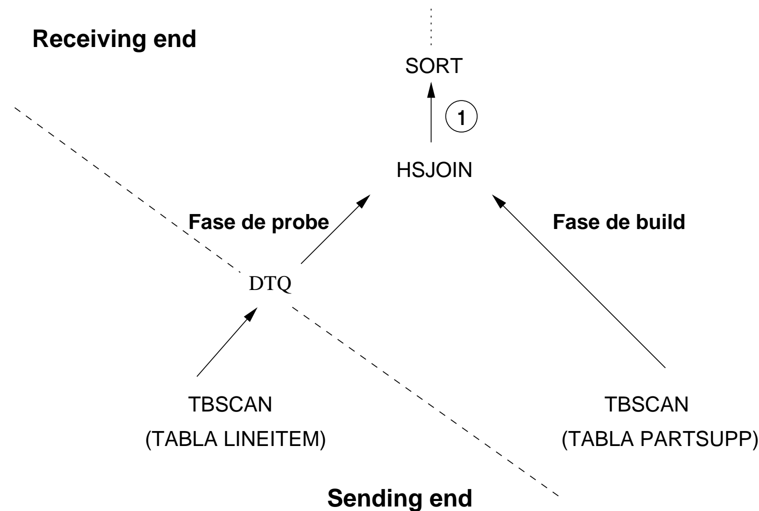


Figura 3.4 Plan de ejecución.

El join entre las relaciones *lineitem* y *partsupp* es ejecutado mediante el operador hash join que implementa el algoritmo *Hybrid Hash Join*. La fase de build puede ser ejecutada de forma local ya que, en este caso, *partsupp* está colocalizada (la tabla ha sido particionada por la clave de join *p_partkey*). Opuestamente, *lineitem* no está colocalizada porque su clave de particionado es *l_orderkey*, y por lo tanto, la tabla $lineitem$ debe de ser reparticionada durante la fase de probe del operador hash join utilizando como calve de particionado *l_partkey*.

Los operadores de reparticionado en DB2 son llamados *Table Queues* [10]. En particular, tal y como se muestra en la Figura 3.4, se utiliza un operador de reparticionado punto a punto (*Direct Table Queue* (DTQ) en DB2), en lugar de una operador de reparticionado que hiciera *broadcast* al resto de los nodos del sistema (*Broadcast Table Queue* (BTQ) en DB2).

Debido a la restricción en *partsupp*, el optimizador de DB2 decide utilizar los *bit filters* durante la operación de join. La selectividad del *bit filter* para cualquier nodo está alrededor de $S_{bf} = 0.55$.

Un aspecto importante de esta consulta es que nos permite simular la ejecución de múltiples planes de ejecución. Con el fin de simular la ejecución de diferentes consultas, introducimos un retraso en tiempo en el operador de reparticionado encargado de recibir datos (*receiving end*). Introduciendo estos retrasos en tiempo por cada tupla recibida en el *receiving end* ($t_{process_{nf}}$ en el modelo analítico), estamos simulando consultas con subplanes de ejecución más complejos por encima de operador de reparticionado. Nuestros resultados sólo varían el tiempo de proceso de una tupla en el *receiving end* porque es suficiente para nuestros objetivos.

Límite del tamaño de grupo para RBF$_R$

En la Sección 3.2, discutimos el valor de GR dependiendo del *heap* de memoria disponible (S_{heap}) y de la capacidad de nuestro sistema de comunicación (C). En este trabajo, limitamos GR basándonos en la capacidad, ya que en nuestro caso, C es más restrictivo que el *heap* de memoria disponible. Así pues, para nuestras pruebas, la capacidad del sistema de comuncicación es $C = 3$ para cada conexión abierta. Como la selectividad del *bit filter* es $S_{bf} = 0.55$, inicializamos $GR = 5$ basándonos en la ecuación 3.1.

El modelo analítico

En el Apéndice A se presenta un modelo analítico para RBF$_R$, RBF$_B$ y la ejecución *baseline*. En la siguiente Sección también se presentan resultados para el modelo analítico que se comparan con los resultados reales con el fin de validar el modelo. La consecuente validación del modelo nos permitirá realizar un análisis comparativo de las tres técnicas para los tres diferentes componentes: el *sending end*, el *receiving end*, y la capa de comunicación.

Para la comparación y la validación del modelo se asume que el *sending end*, *receiving end* y la capa de comunicación se ejecutan en paralelo. De este modo, el componente de coste más elevado para cualquiera de las tres estrategias, determina el tiempo de ejecución de la estrategia.

En la práctica, DB2 ejecuta diferentes procesos para los *sending end* y receving end de la capa de comunicación [10]. Para la consulta que estamos ejecutando, los procesos *sending end* y *receiving end*, junto al proceso encargado de la capa de comunicación son los que más carga tienen, y asumimos que los tres componentes son ejecutados en paralelo al tener 4 procesadores disponibles en cada nodo. Sin embargo, este no sería el caso cuando más de una consulta se ejecuta en paralelo. En este caso, el número de procesos sería mucho mayor que el número de procesadores físicos, de modo que no se podría asumir ningún solapamiento. En consecuencia, sólo se pueden esperar resultados precisos de nuestro modelo analítico cuando validemos nuestro modelo para el caso de una sola consulta. Sin embargo, aunque menos preciso, nuestro modelo podría ser también válido para casos en los que el número de procesos es mayor que el número de procesadores.

3.4 Resultados

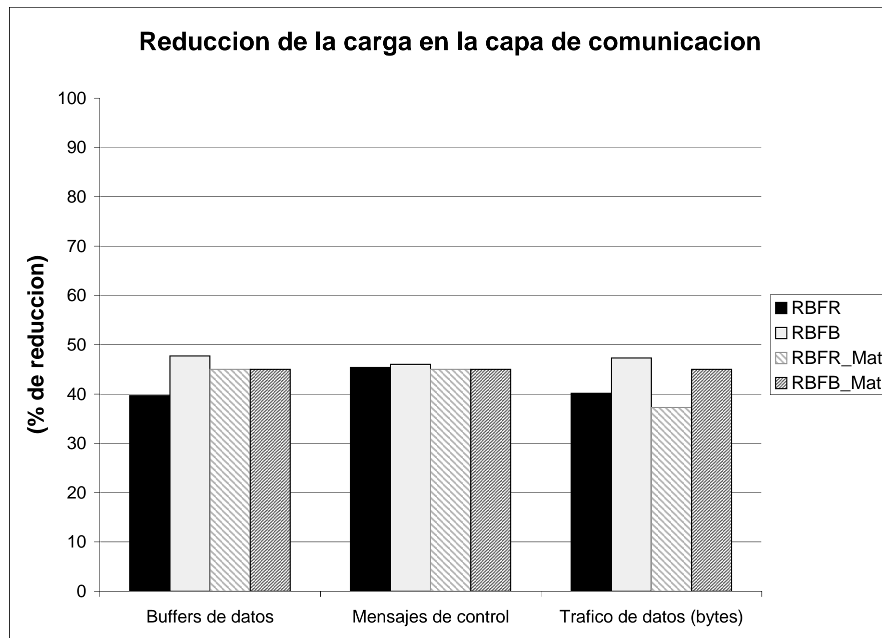
En esta Sección, basándonos en los resultados obtenidos a través de las ejecuciones reales, evaluamos y comparamos la ejecución *baseline*, los *Remote Bit Filters Broadcasted* (RBF$_B$), y nuestra propuesta, los *Remote Bit filters with Requests* (RBF$_R$).

Realizamos nuestra evaluación dentro de dos contextos distintos: la ejecución aislada de una consulta, y la ejecución de varias consultas de forma concurrente. Esto nos dará información importante relativa a dos aspectos importantes: (i) cómo la consulta se beneficia de nuestra estrategia, y (ii) cómo se reduce la presión en la capa de comunicación, viéndose beneficada la ejecución en paralelo de múltiples consultas.

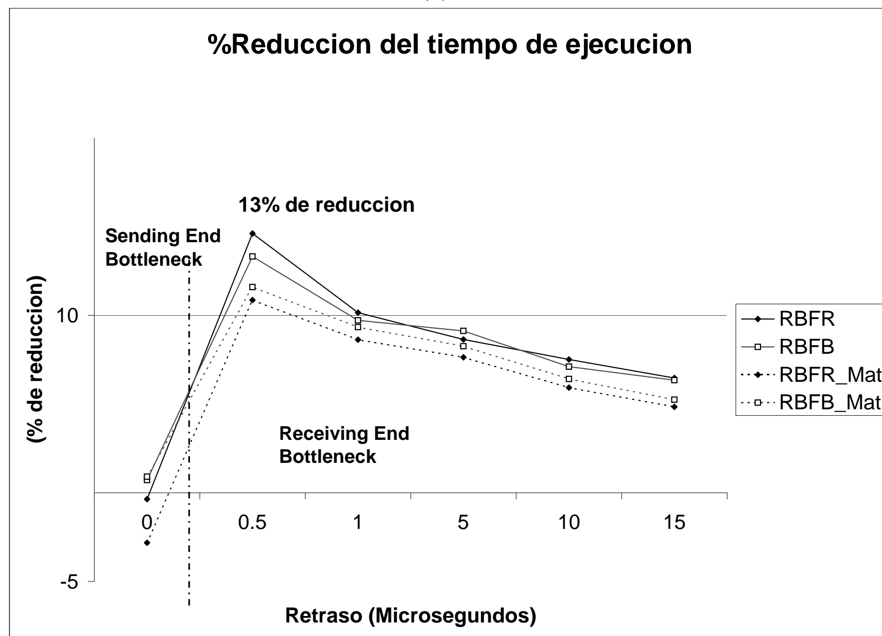
También, validaremos el modelo analítico propuesto en el Apéndice A comparando los tiempos reales de ejecución con los obtenidos por el modelo analítico. Los resultados muestran que el modelo sigue las tendencias de las ejecuciones reales.

Ejecución secuencial

Empezamos analizando la influencia de cada estrategia en la capa de comunicación. Lo hacemos con la ayuda del gráfico mostrado en la Figura 3.5-(a). En ese gráfico se muestra la reducción del número de *buffers* de datos, *buffers* de control y del tráfico de datos, por las técnicas RBF$_R$ y



(a)



(b)

Figura 3.5 Ejecución secuencial. Porcentajes de reducción de RBF_R y RBF_B respecto la ejecución *baseline*. (a) Porcentaje de reducción en la carga de la capa de comuncicación; (b) Porcentaje de reduccion en el tiempo de ejecución.

RBF_B respecto la ejecución *baseline*. Se muestran resultados para las ejecuciones reales (RBFR y RBFB)y para nuestro modelo (RBFR_Mat y RBFB_Mat).

En el gráfico, podemos observar que el número de *buffers* de datos se reduce de forma considerable: 47% para RBF$_B$ y un 40% para RBF$_R$. Estos resultados se explican a partir de la eficiencia de los *bit filters* de un 45% ($E_{bf} = 1 - S_{bf} = 0.45$). Era de esperar el hecho de que RBF$_R$ no obtenga unos resultados tan óptimos como los obtenidos por RBF$_B$. Esto es debido al mecanismo utilizado por ambos al empaquetar las tuplas. RBF$_B$ chequea localmente en cada nodo el código de hash de cada tupla en el correspondiente *bit filter* del nodo destino, y empaqueta la tupla en el *buffer* en caso de no ser filtrada. RBF$_R$ se comporta de forma diferente: se prepara un grupo de GR *buffers* de datos, y sus códigos de hash son enviados al *receiving end*. Entonces, el *sending end* recibe un *bitmap* representando el grupo de *buffers*. Finalmente, basándose en el *bitmap* recibido, un número de tuplas es eliminado de los GR *buffers* de datos, lo cual conlleva a un número más pequeño de tuplas enviadas, y probablemente, a la creación de *buffers* de datos parcialmente llenos. El número de *buffers* parcialmente llenos y enviados, incrementa levemente el número total de mensajes comparado con RBF$_B$.

El número de mensajes de control también es levemente mayor para RBF$_R$ que para RBF$_B$. En este caso, la razón está en que los mensajes de control se envían cada C mensajes, tal y como se explicó en la Sección 3.3. Con esto, dado que hay muchos más mensajes de datos, también debe de haber un incremento en el número de mensajes de control.

Esta reducción es significativamente importante ya que puede ser vista como una clara mejora de la sincronización entre el *receiving end* y *sending end* del operador de reparticionado.

El resultado de mayor impacto es la reducción del tráfico de datos que obtenemos para RBF$_B$ y RBF$_R$, que siguiendo la línia de los anteriores resultados es de un 47% y 40% respectivamente. El aspecto más importante es que, introduciendo los parámetros de DB2 en las ecuaciones 3.2 y 3.4, la cantidad de memoria necesitada por el *heap* de RBF$_R$ es más de 20 veces menor que el necesitado por RBF$_B$.

Tiempo de ejecución de la consulta. Hasta ahora, hemos visto que la comunicación de datos ha sido reducida de forma significativa con el uso de *Remote Bit Filters*. Ahora nos gustaría entender si a su vez tiene un impacto colateral en el tiempo de ejecución de una consulta.

La Figura 3.5-(b) muestra un gráfico con el porcentaje de mejora en tiempo de ejecución por RBF$_R$ y RBF$_B$ respecto la ejecución del código original (ejecución *baseline*). Mostramos resultados para ejecuciones actuales (lineas sólidas) y para los resultados estimados por el modelo analítico (lineas discontinuas). El eje de ordenadas del gráfico muestra la influencia de los diferentes tiempos de proceso aplicados a las tuplas recibidas por el *receiving end*, $t_{process_{nf}}$ en el modelo analítico.

Los valores de las variables del modelo analítico utilizados para obtener los gráficos de la Figura 3.5-(b) se han obtenido directamente de los datos observados en el optimizador de DB2, en caso de que estuvieran disponibles, y a partir de medidas reales en caso contrario.

Empecemos por justificar por qué la capa de comunicación es el componente más costoso en este caso. Inicializando el tiempo de procesado de una tupla en el *receiving end* a cero (Figura 3.5-(b)), podemos ver que, de nuevo, el beneficio tanto de RBF$_R$ como RBF$_B$ es imperceptible. Si la capa de comunicación fuera el componente más cotoso, entonces, sí que obtendríamos alguna mejora. Al no ser éste el caso, la razón viene del hecho que tanto el *sending end* como el *receiving end* son el cuello de botella de la consulta. A través del modelo analítico se puede apreciar que el *receiving end* mejora en tiempo de ejecución tanto para RBF$_R$ como RBF$_B$. Así concluimos, que para un retraso de procesado de tupla igual a cero, el *sending end* es el componente más costoso.

Veamos que ocurre cuando incrementamos el tiempo de proceso de una tupla en el *receiving end*, de modo que éste pasa a ser el componente más costoso. Empezando por un tiempo de procesado de 0.5μsegundos, obtenemos una máxima mejora del 13%. Con tiempos de procesado

mayores, la mejora disminuye. Remitiéndonos al modelo analítico, estamos añadiendo tiempo de procesado a las tuplas que no han sido filtradas (incrementando $t_{process_{nf}}$), de modo que, el tiempo de computación ahorrado por el uso tanto de RBF$_R$ como de RBF$_B$ es más pequeño comparado con el tiempo total de ejecución.

Consultas ejecutadas de forma concurrente

En esta Sección analizamos cómo la presión en la capa de comunicación influye en el tiempo de ejecución de las consultas ejecutadas de forma concurrente. Para ello, empezamos ejecutando la misma consulta dos, tres, y cuatro veces de forma concurrente.

En este caso, la capa de comunicación debe de ser el cuello de botella y pasa a tener un papel determinante en el tiempo total de ejecución. La razón es que la red es un solo recurso compartido por múltiples operadores de reparticionado (en este caso uno por consulta), que cargan la red en paralelo.

Con estos experimentos estamos testeando el efecto de nuestra estrategia sobre la capa de comunicación. Por esta razón, ningún tipo de retraso es aplicado al procesado de cada tupla recibida en el *receiving end*.

Aunque el modelo analítico no aplica en este caso debido a que no asume solapamiento de ningún tipo entre procesos, mostramos los resultados del modelo asumiendo un proceso por procesador.

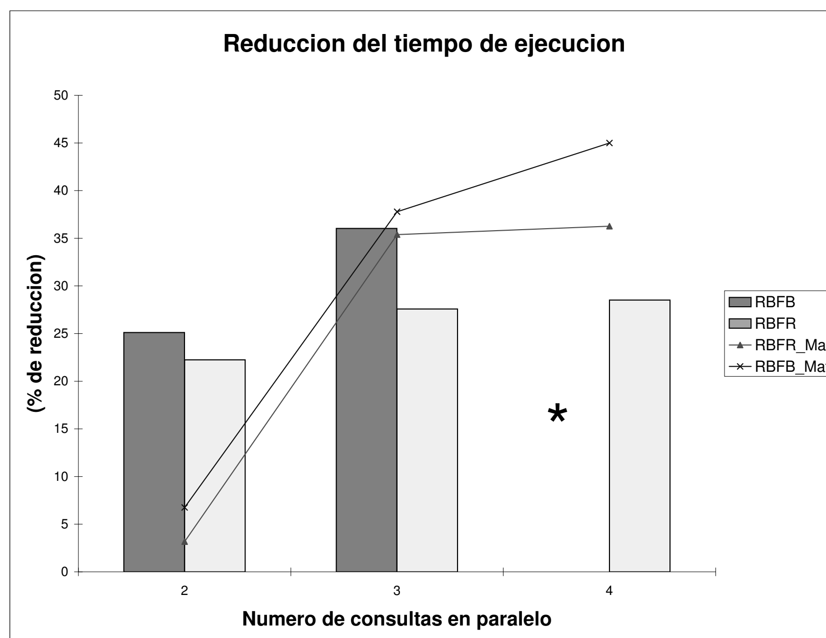


Figura 3.6 Reducción del tiempo de ejecución de RBF$_R$ y RBF$_B$ respecto a la ejecución *baseline* cuando ejecutamos varias consultas de forma concurrente. Las lineas de barras son los resultados de las ejecuciones reales. Las lineas sólidas son los resultados obtenidos por el modelo analítico. $*$ significa que las consultas no pudieron ser ejecutadas debido a falta de memoria.

La Figura 3.6 muestra el porcentaje de tiempo de ejecución reducido por cada ejecución con dos, tres, y cuatro consultas ejecutadas en paralelo. El tiempo de ejecución mejora de forma significativa respecto a la ejecución original, siempre sobre un 20% para RBF$_R$ y hasta un 30% para RBF$_B$ para

dos y tres consultas en paralelo. En este caso, el decremento de datos comunicados a través de la red de comunicación queda bien reflejado en el tiempo total de ejecución.

El gráfico también muestra que RBF_B supera RBF_R: tiene sentido desde el punto de vista que RBF_B añade menos tráfico a la red. Sin embargo, un aspecto importante es que para la prueba en el que se ejecutan cuatro consultas de forma concurrente, mientras RBF_B no puede ser ejecutado debido a falta de memoria, RBF_R al requerir menos recursos de memoria, puede ser utilizado y obtiene una mejora del 27%. También, mostramos resultados para el modelo analítico en lineas continuas. Podemos ver que para este caso, el modelo analítico falla para el caso de dos consultas, sin embargo es algo más preciso para el caso de tres y cuatro consultas. En la Figura 3.7 se muestra la gráfica relacionada con el comportamiento de la capa de comunicación. La gráfica muestra el porcentaje de reducción respecto el total de tráfico de datos comunicados. Las gráficas relativas al porcentaje de reducción respecto al número de *buffers* de datos y *buffers* de control no se muestran por su similitud con el presente.

Los resultados en este caso son similares al caso de la ejecución con una sola consulta. Las mejoras sobre la ejecución *baseline* con el solo uso de los *bit filters* de forma local, son similares sin importar el número de consultas ejecutadas. Los porcentajes de reducción están en torno 47% para RBF_B y en torno el 40% para RBF_R respecto al número de *buffers*, tanto de datos como de control enviados, y respecto del total de tráfico de datos en la red.

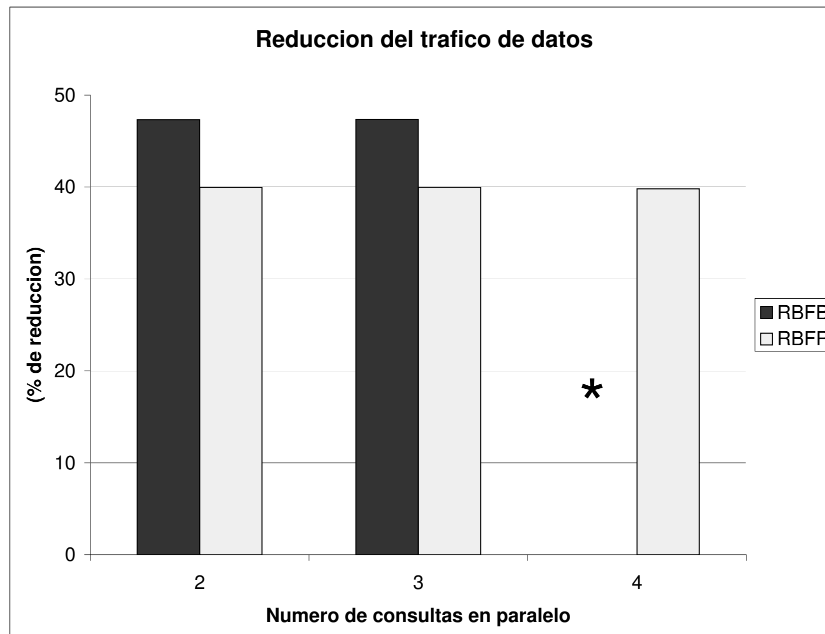


Figura 3.7 Reducción de carga en la capa de comunicación de RBF_R y RBF_B respecto la ejecución *baseline*. $*$ significa que las consultas no pudieron ser ejecutadas debido a falta de memoria.

3.5 Análisis comparativo

Una vez mostrados los resultados y basándonos en el modelo analítico propuesto en el Apéndice A, y validado en la Sección anterior, queremos comparar las tres estrategias para entender cuando una es mejor que otra. El coste final de cada estrategia es muy dependiente de las características del

SGBD y de la arquitectura en la que se ejecuta. Sin embargo, el modelo que hemos propuesto es fácilmente generalizable y se asume que tiene en cuenta tanto el SGBD como la arquitectura utilizadas.

Para una mejor comprensión del análisis comparativo que exponemos a continuación, se recomienda la lectura previa del del modelo analítico presentado en el Apéndice A.

Comparativa entre componentes

Queremos entender las diferencias entre los componentes para cada estrategia. Así pues, comparamos el *sending end*, *receiving end* y la capa de comunciación para las tres estrategias: RBF$_R$, RBF$_B$, y la ejecución *baseline*. Esta comparación es posible porque las variables para cada componente en las diferentes estrategias son similares.

Sending end Podemos decir que RBF$_R$ es el componente de más coste durante la ejecución del *sending end*. De hecho, $t_{write_{RBFR}}$ es siempre mayor que $t_{write_{baseline}}$ (restamos la ecuación A.1 de la ecuación A.2).

También, $t_{write_{RBFR}} > t_{write_{RBFB}}$: obtenemos $N\ (t_{pack_{hc}} + t_{filter}E_{bf} + t_{pack}E_{bf} - t_{test})$ de la restamos la ecuación A.3 de la ecuación A.2). Podemos asumir esto si t_{test} es menor que la resta de las variables en la ecuación, lo cual es factible en cualquier contexto.

Finalmente, la diferencia entre RBF$_B$ y la implementación *baseline* es $N(t_{test} + t_{hc} - t_{pack}E_{bf})$ si restamos la ecuación A.3 de la ecuación A.1. Así pues, concluimos que RBF$_B$ reduce el coste del *sending end* sólo cuando $\frac{t_{test}+t_{hc}}{tpack} < E_{bf}$.

Receiving end La implementación *baseline* es, en este caso, la de más coste (restamos la ecuación A.5 de las ecuaciones A.4, y A.6).

RBF$_R$ es más costoso que RBF$_R$ en $N(t_{unpack_{hc}} + t_{test})$. Sin embargo, hemos observado que la diferencia entre los dos métodos no es significante cuando el *receiving end* es el componente de más coste. Como se demostrará más tarde, esto sucede debido a que desempaquetar los códigos de hash y testearlos en el *bit filter* tiene un coste muy pequeño.

Capa de comunicación La comparación del modelo para la capa de comunicación resulta un poco más compleja. Sin embargo, podemos simplificar el proceso si primero definimos $|B_{hc}|$ y $|B_{be}|$ en términos de $|B_d|$. Para ello, definimos S_t como el tamaño de una tupla una vez empaquetada en un *buffer* de datos, y S_{hc} como el tamaño de un código de hash una vez empaquetado en un *buffer* de códigos de hash. Así pues, en el espacio que ocupa una tupla en un *buffer* de datos, podemos almacenar un total de $x = \frac{S_t}{S_{hc}}$ códigos de hash. De esta forma, podemos redefinir $|B_{hc}|$ como $x|B_d|$. También, si b es el número de bits ocupado por un código de hash, tenemos que $|B_{be}| = bx|B_d|$. Después de estas redefiniciones podemos proceder a la comparación de las tres estrategias.

Comparativa entre técnicas

Empecemos comparando la estrategia *baseline* con RBF$_R$ y RBF$_B$:

- RBF$_R$. Si restamos la ecuación A.8 de la ecuación A.7, tenemos que la diferencia es:

$$\frac{N\lambda}{B_d}\left[\left(E_{bf} - (\frac{1}{x} + \frac{1}{bx})\right) B_{d_s} + \frac{E_{b_f}}{C} B_{c_s}\right]$$

Asumiendo que B_{d_s} es mucho mayor que B_{h_s} y que B_{c_s}, entonces podemos decir que $t_{comm_{RBFR}}$ reduce el coste de la capa de comunicación si $E_{bf} > \frac{1}{x}\left(1 + \frac{1}{b}\right)$. Como $b \gg 1$, podemos simplificar la expresión a $E_{bf} > \frac{1}{x}$. Normalmente E_{bf} es mayor que 0.5, queriendo decir que con el simple hecho de que S_{hc} sea el doble que S_t la técnica es beneficiosa. Para casos en los que $E_{bf} \simeq \frac{1}{x}$, deberíamos tener en cuenta lo mesajes de control salvados.

- RBF$_B$. Si restamos la ecuación A.9 de la ecuación A.7 podemos ver que RBF$_B$ siempre reduce el coste en:

$$\frac{NE_{bf}\lambda}{|B_d|}\left[B_{d_s} + \frac{1}{C}B_{c_s}\right]$$

Como pasaba con el *receiving end*, podemos decir que tanto RBF$_R$ como RBF$_B$ mejoran el rendimiento del SGBD si la capa de comunicación es el componente de más coste.

A continuación estudiamos bajo que circunstancias RBF$_R$ se asemeja a RBF$_B$ respecto a la reducción del tráfico de datos en la capa de comunicación.

En la Figura 3.8 ilustramos, variando la eficiencia del *bit filter* $(E_{bf} = 1 - S_{bf})$, y para diferentes valores de x, el porcentaje de diferencia entre RBF$_B$ y RBF$_R$ en términos de comunicación de datos en la red. Para $x \geq 4$ y $E_{bf} \geq 0.5$, la diferencia entre RBF$_R$ y RBF$_B$, es menor o igual que el 5%. Valores para $x \geq 4$ son razonables teniendo en cuenta que el tamaño de una tupla S_t es normalmente mayor que el tamaño de un código de hash S_{hc}. Valores para $E_{bf} \geq 0.5$ también son razonables ya que el uso de *bit filters* con eficiencias menores a un 0.5 serían descartados por el optimizador.

Por lo tanto concluimos que, exceptuando unos pocos casos en los que el tamaño de la tupla es pequeña y la selectividad del *bit filter* es alta, RBF$_R$ se comporta de forma similar a RBF$_B$.

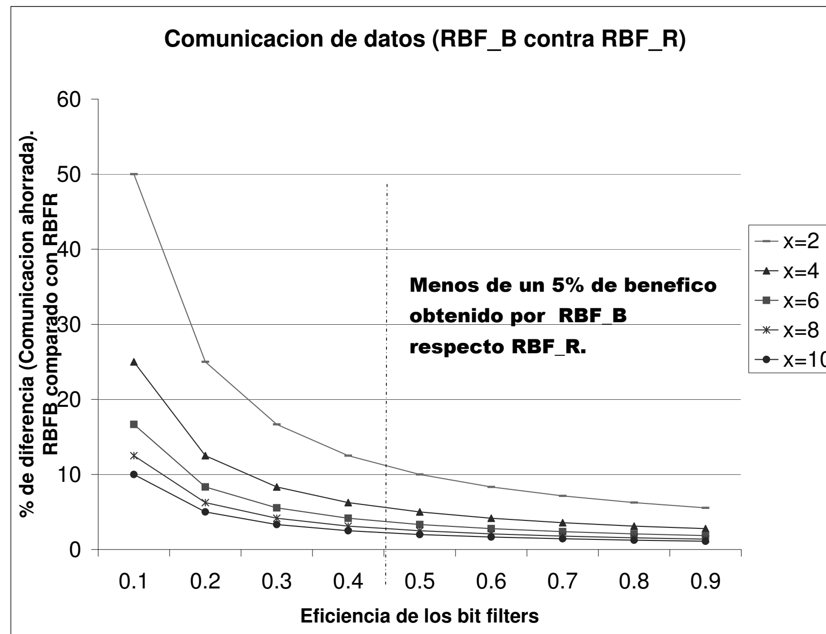


Figura 3.8 Diferencia entre RBF$_B$ y RBF$_R$ en términos del porcentaje ahorrado en comunicación de datos. x es la relación entre el tamaño de la tupla y el tamaño de un código de hash.

Recursos de Memoria (RBF$_R$ y RBF$_B$) . Recordemos que la cantidad de memoria utilizada por RBF$_B$ depende de forma exclusica del tamaño de los *bit filters* creados en cada nodo. Inicializando los valores $d = 1$ y $P = 0.05$ asociados a un bit, entonces el tamaño de un *bit filter* depende del número de valores distintos n provenientes de la relación de build, que bajo las premisas del modelo analítico, es el mismo para cualquier nodo.

La Figura 3.9 muestra, para diferentes grupos de valores distintos n, el ahorro que RBF$_R$ obtiene comparado con RBF$_B$. La memoria usada por RBF$_R$ ha sido calculada tomando la media de distintos valores de x ($x = \frac{S_t}{S_{hc}}$), sobre un rango de 2 a 10. También tomamos el valor óptimo para GR (ver ecuación 3.3 de la Sección 3.3). De modo que, cuando la efectividad de un *bit filter* es alta, RBF$_R$ requiere un alto número de *buffers* por cada nodo destino.

En la gráfica podemos apreciar que en la mayoría de los casos, incluso para un reducido número de valores distintos ($n = 10^6$), RBF$_R$ tan solo utiliza un 20%, o menos, de los recursos de memoria empleados por RBF$_B$. Valores cercanos a $n = 10^6$ son relativamente pequeño en entornos *DataWarehouse* cuyas tendencias conducen a volúmenes de datos entorno a los Petabytes de información. A modo de ejemplo, para un TPC-H de 100GB, la relación $orders$ tiene un número de valores distintos de $n = 15 \times 10^7$, de modo que, sólo una selectividad de 0.007 podria restringir $orders$ a 10^6 valores distintos. Sólo en casos extremos, para conjuntos de datos con pocos valores distintos ($n = 10^5$), donde los *bit filters* son muy pequeños en tamaño y tienen una alta efectividad, entonces, RBF$_B$ consumiría menos recursos de memoria que RBF$_R$.

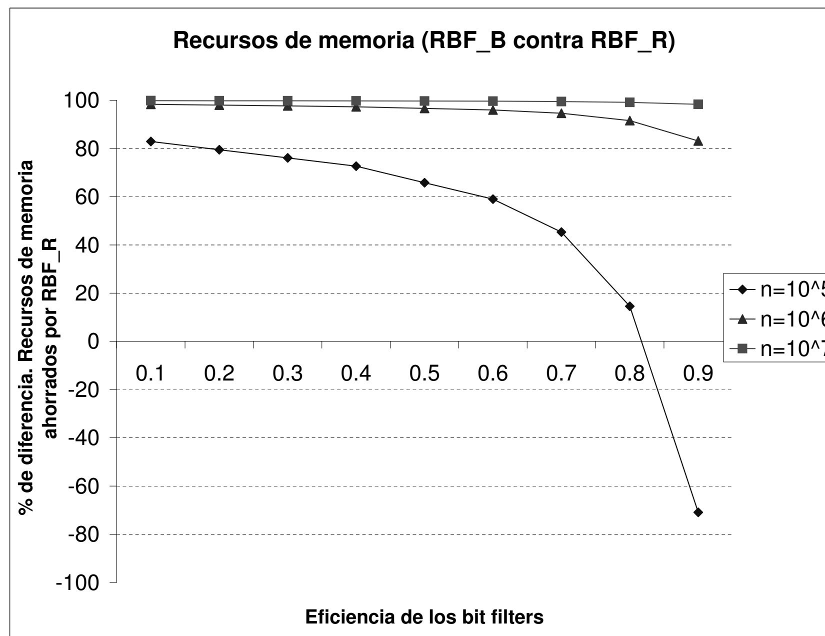


Figura 3.9 Diferencia entre RBF$_B$ y RBF$_R$ en términos de utilización de los recursos de memoria. n es el número de valores distintos provenientes de la relacion de build.

A partir de los resultados y del análisis comparativo de los modelos, podemos concluir que, excepto en el caso en el que el *sending end* es el comopnenete más caro, en el resto de los casos, el uso de los *Remote Bit Filters* mejoraría el rendimiento del SGBDs. En estos casos, la diferencia

entre RBF_R y RBF_B no es significativa, lo que permite decantarnos en favor de RBF_R debido al reducido uso de los recursos de memoria si lo comparamos con RBF_B.

3.6 Conclusiones

La propuesta expuesta de *Remote Bit Filters with Requests* (RBF_R), prueba que es posible utilizar los *bit filters* en la ejecución de joins paralelos no colocalizados, reduciendo la comuncicación de tuplas en la red, y con un bajo consumo de los recursos de memoria.

En particular, hemos comparado nuestra propuesta con otras técnicas ya implementadas (el uso local de los *bit filters*), o ya mencionadas en la literatura pero no implementadas en un SGBD, los *Remote Bit Filters Broadcasted*, RBF_B. Para nuestro análisis comparativo se han implementado dos protótipos sobre el SGBD de IBM (DB2 UDB v8.1), junto a los modelos analíticos correspondientes. Para la evaluación se ha utilizado una arquitectura cluster formada por 8 nodos con una configuración SMP cada uno.

Los aspectos mas importantes a destacar de nuestra técnica son:

- RBF_R minimiza el uso de la memoria comparado con RBF_B, además, mostrando un rendimiento muy similar. Por ejemplo, para la ejecución de varias consultas en paralelo, siendo el consumo de memoria de RBF_R más de 20 veces inferior al de RBF_B, obtenemos que el tráfico de datos en la red de interconexión se reduce en un 40% y un 47% para RBF_R y RBF_B respectivamente.

- Ahorrando recursos de memoria permitimos una ejecución más rápida de otras operaciones, mejorando así el rendimiento global del SGBD. En nuestras pruebas, la ejecución en paralelo con 4 consultas no pudo ser ejecutada por RBF_B debido al agotamiento de memoria, mientras que RBF_R sí pudo utilizarse obteniendo una mejora del 27% en el tiempo de ejecución.

- Como se muestra en nuestro modelo, y a través de las ejecuciones reales de nuestro prototipo, la carga de mensajes introducida por RBF_R no degrada el rendimiento del SGBD.

- En términos de comunicación de datos, nuestros modelos muestran que la diferencia entre RBF_R y las técnicas previas, se reduce a medida que la eficiencia del *bit filter* y el tamaño de las tuplas incrementa. Los resultados muestran que utilizando *bit filters* con una eficiencia superior o igual a 0.4, la diferencia entre RBF_R y RBF_B es siempre inferior al 10%, reduciéndose esta diferencia cuanto mayor es el tamaño de las tuplas comunicadas.

4

PUSHED DOWN BIT FILTERS

4.1 Introducción

La E/S resulta determinante en el tiempo de ejecución de consultas que requieren el procesado de grandes volúmenes de datos, como consultas de tipo join estrella donde los datos de la tabla de hechos son relacionados con las tablas que la dimensionan, y agrupados con fines analíticos. En estos casos, la E/S producida por la cantidad de datos a procesar de la tabla de hechos se convierte en un serio cuello de botella.

En el Capítulo anterior hemos presentado los *Remote Bit Filters*, una estrategia que, mediante el uso remoto de los *bit filters*, ahorra comunicación de datos entre los nodos en sistemas paralelos distribuidos. En este Capítulo nos centramos en el uso de los *bit filters* con el fin de ahorrar procesamiento de datos durante la ejecución de consultas sobre grandes bases de datos relacionales, tanto en sistemas secuenciales, como en sistemas paralelos con recursos compartidos.

4.1.1 Trabajo relacionado y motivación

Los *bit filters* [14] han sido utilizados para reducir la E/S sobre consultas con múltiples join [46] [49]. Sistemas Gestores de Bases de Datos comerciales como DB2 UDB [46], Oracle [42] y Microsoft SQL Server [46], utilizan *bit filters* con el fin de reducir la E/S durante las operaciones de join, como en el *Hybrid Hash Join* [77]. En estos SGBDs, y como se explicó en el Capítulo 2, los *bit filters* son construídos durante la fase de *build* de la operación hash join. Entonces, durante la fase de *probe* , son utilizados para filtrar tuplas de la relación de *probe* . Este uso de los *bit filters* reduce la E/S generada por los propios operadores hash join [77].

En este Capítulo vamos más allá del uso corriente de los *bit filters*, y proponemos *Pushed Down Bit Filters* (PDBF), una nueva estrategia que se basa en el uso de los *bit filters*, creados localmente por los operadores de join, en los nodos *scan* del plan de ejecución. El fin de la técnica reside en reducir el volumen de datos a procesar entre los nodos intermedios de un plan de ejecución. En consecuencia, se reduce el tiempo de proceso, y en la mayoría de casos, la E/S también se ve reducida.

El trabajo propuesto en [26] utiliza los *bit filters* de forma similar, con el objetivo de ahorrar tráfico entre los nodos del plan disminuyendo la cantidad de E/S y el tiempo de cpu. Los resultados obtenidos son realmente interesantes y tienen un par de propiedades de interés que influyen en el tiempo de ejecución de una consulta. Primero, los nuevos *bit filters* son creados cuando es posible

en cada nodo del plan de ejecución. Segundo, los nodos del plan tienen que ser ejecutados de forma secuencial, opuesto a la usual ejecución segmentada del plan.

En contraste, PDBF se adapata a la ejecución segmentada del plan. Tal y como se demuestra en [82], la ejecución segmentada deriva a un mejor rendimiento para la ejecución de consultas con múltiples joins, sugiriendo que algoritmos segmentados deberían de ser usados en sistemas de flujos de datos. Además, la mayoría de SGBDs comerciales, así como DB2 UDB u Oracle [76], implementan la filosofía segmentada durante la ejecución del plan de ejecución. Otra diferencia importante respecto a la propuesta en [26], es que PDBF utiliza los *bit filters* ya creados por defecto en las operaciones de join, y por lo tanto no incurre en ningún coste extra de cpu o espacio de memoria. Sin embargo, nuestra estrategia sólo puede ser usada cuando los join que crean los *bit filters*, como el hash join, aparece en el plan de ejecución.

A parte de las características mencionadas anteriormente, PDBF presenta algunas propiedades adicionales de interés:

- La cantidad de meta-datos necesarios para implementar la estrategia es pequeña y puede ser fácilmente utilizada por el motor del SGBD.
- Su implementación es relativamente simple.
- Podemos reducir casi al mínimo la cantidad de E/S a realizar por el plan de ejecución.
- PDBF reduce significativamente el tiempo de proceso por parte de la cpu.
- No incrementamos el tiempo de ejecución en las consultas en las que la técnica no es efectiva.

Con el fin de reafirmar las propiedades mencionadas, durante el resto del Capítulo analizamos con detalle nuestra propuesta. Explicamos el contexto bajo el cual las consultas pueden beneficiarse de nuestra estrategia, su aplicabilidad, y cuales son las estructuras necesarias a mantener por el motor del SGBD.

4.1.2 Contribuciones

Proponemos *Pushed Down Bit Filters* (PDBF), una técnica que explota el uso de los *bit filters* con el fin de ahorrar tráfico entre los nodos del plan de ejecución de una consulta.

Implementamos y evaluamos PDBF sobre el SGBD PostgreSQL [70] bajo una base de datos TPC-H [2] de 1GB. Medimos la cantidad de tráfico de datos intra-operadores en el plan de ejecución. También medimos la E/S y el tiempo de ejecución para diferentes *heaps* de memoria. Mientras el tráfico de datos entre los nodos del plan es independiente del SGBD, la E/S depende de la cantidad de memoria disponible para ejecutar la consulta, y el tiempo de ejecución depende de la plataforma utilizada y del SGBD. Las medidas del tráfico de datos y de la E/S nos ayudan a entender los beneficios de forma genérica, mientras que el tiempo de ejecución ayuda a entender los beneficios bajo una plataforma y SGBD concretos.

Comparamos PDBF con la ejecución original en PostgreSQL y el uso clásico de los *bit filters*. Los resultados muestran que nuestra estrategia reduce el tiempo de ejecución y la E/S de algunas consultas en un 50%.

4.2 Pushed Down Bit Filters (PDBF)

El objetivo de PDBF, es obtener el máximo beneficio posible de los *bit filters* creados en los operadores de join de una consulta. En este Capítulo nos centramos en los *bit filters* creados

durante la operación de *Hybrid Hash Join*, aunque nuestra estrategia podría ser extendida a otras operaciones de join.

La idea detrás de los PDBF es muy simple, y consiste en utilizar los *bit filters* generados en los nodos hash join del plan de ejecución, en los nodos inferiores del plan. Esto será posible en aquellos nodos hoja que *scaneen* una relación que contenga atributos sobre los que, los *bit filters* ya generados, puedan ser aplicados. Incluso relaciones que contengan atributos que tienen dependencias no funcionales con el atributo utilizado para construir el bit filter, pueden ser filtrados.

Esta eliminación prematura de tuplas tiene como objetivo reducir el volumen de datos a procesar por el plan de ejecución y por lo tanto una disminución del trabajo de los nodos que lo componen.

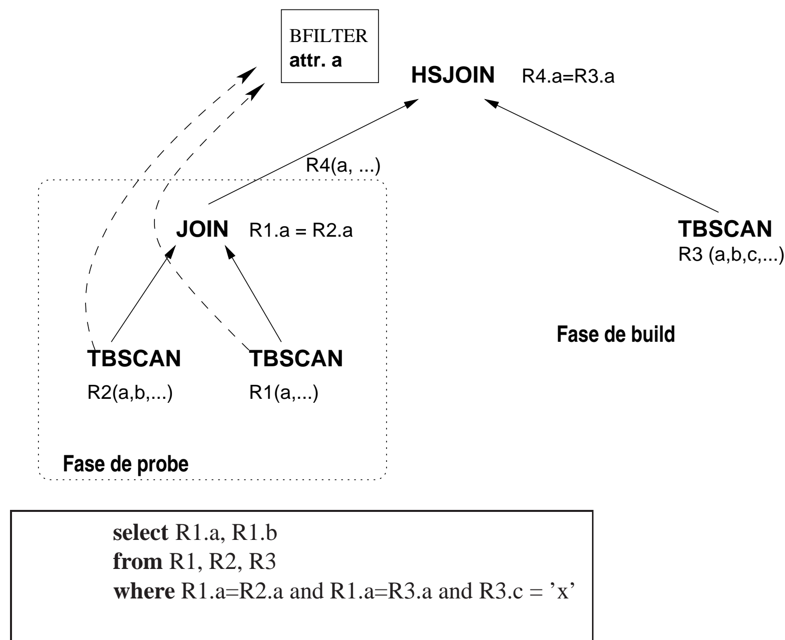


Figura 4.1 Ejemplo de la técnica PDBF.

La Figura 4.1 muestra un ejemplo sencillo de la técnica. El nodo hash join, durante la fase de *build* , crea un *bit filter* sobre el atributo *a* de R3. Una vez terminada la fase de *build* , se ejecuta el subárbol que conforma la fase de *probe* . Este subárbol está compuesto por una operación join cualquiera sobre el atributo *b*, y dos nodos *scan*. Cuando el subárbol se ejecute para ir suministrando tuplas al nodo hash join, los nodos *scan* chequearán las tuplas contra el bit filter, ya que tanto R1 como R2 poseen el atributo *a* sobre el cual ha sido creado. De esta forma descartamos tuplas antes de que lleguen al nodo hash join, reduciendo así, el volumen de datos a procesar por el join y disminuyendo su cardinalidad de salida aminorando el coste del hash join.

Nuestra estrategia no requiere la creación adicional de ninguna estructura de datos especial. La única información necesaria en los nodos *scan*, es una estructura que relacione cada atributo con el *bit filter* relacionado.

4.2.1 Contexto de ejecución

Debido a caracterícas especiales de los *bit filters* y planes de ejecución, hay ciertas restricciones a nuestra estrategia:

- Un nodo *scan* sólo puede consultar los *bit filters* que hayan sido creados en operaciones hash join que pertenezcan al mismo *select statement* dentro de una consulta. Cada *select statement* tiene flujos de datos que no interfieren entre si, así que, dentro de un mismo *select statement* no se puede eliminar tuplas a través de la información proveniente de otro *select statement*.

- Un *bit filter* sólo puede ser utilizado por los nodos hoja del subárbol que forma parte de la fase de *probe* del nodo hash join que ha creado dicho bit filter. Un *bit filter* no está completamente creado hasta que la fase de *build* finaliza, así pues, ningún nodo que no pertenezca a la fase de *probe* podrá consultar el bit filter.

- Si un nodo *scan* tiene la posibilidad de actuar sobre un *bit filter* que ha sido creado en una operación hash join que lleva a cabo un *outer join*, entonces, no se puede aplicar la técnica sobre este bit filter. Esto es debido a que cuando un nodo hash join lleva a cabo un *outer join*, los *bit filters* son utilizados, no para eliminar tuplas, sino para proyectarlas si sabemos de antemano que no harán join con ninguna otra tupla de la otra relación.

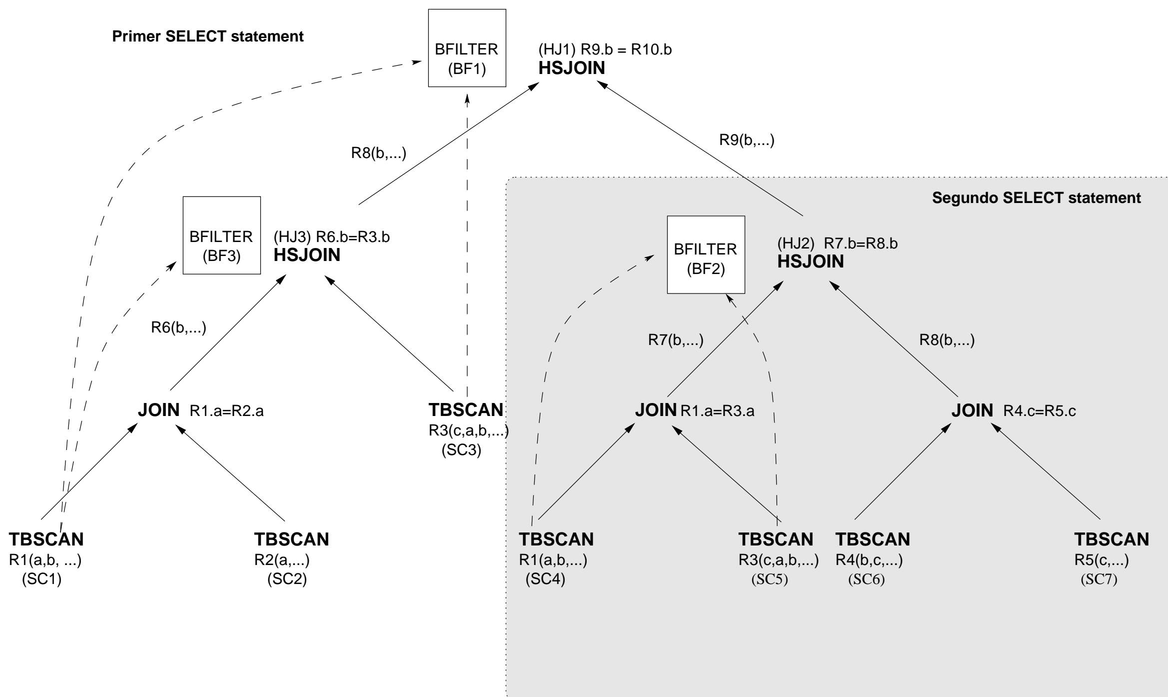


Figura 4.2 Aplicabilidad de los PDBF.

En la figura 4.2 se muestra un plan de ejecución algo complejo con el fin de estudiar la aplicabilidad de la técnica. El plan de ejecución está formado por dos *select statements*. Los *bit filters* de

cada nodo hash join se crean sobre el atributo de join. Como se puede apreciar, el uso de la técnica es local a cada *select statement*:

- **Primer select statement** : los *bit filters* BF1 y BF3 son creados a partir del atributo *b* por los nodos HJ1 y HJ3 respectivamente. Los nodos *scan* que actúan sobre una relación que contenga el atributo *b* son SC1 y SC3.
  - SC1 : podrá consultar tanto BF1 como BF3 ya que pertenece a los subárboles de la fase de *probe* tanto de HJ1 como de HJ3.
  - SC3: pertenece al subárbol de la fase de *probe* de HJ1, pero no de HJ3, así que sólo podrá consultar BF1.
- **Segundo select statement** : el *bit filter* BF2 es creado por HJ2 sobre el atributo de join *b*. Los nodos *scan* que actúan sobre una relación que contenga el atributo *b* son SC4, SC5 y SC6. Los únicos de entre éstos que pertenecen al subárbol de la fase de *probe* de HJ2 son SC4 y SC5. Así pues, aplicaríamos la técnica sobre estos dos nodos *scan*.

4.2.2 El algoritmo

Una vez se haya decidido sobre qué nodos *scan* es aplicable la técnica, es necesario que éstos mantengan la información relativa a los *bit filters* que deben consultar y, sobre qué atributo/s actúan cada uno. Definimos el conjunto S de pares $\{bitfilter, atributo\}$ cómo los datos utilizados por un nodo *scan* para aplicar PDBF. Retomando el ejemplo de la Figura 4.2, el nodo SC1 mantiene la información del conjunto $S = \{\{BF1, b\}, \{BF3, b\}\}$, SC3 la del conjunto $S = \{\{BF1, b\}\}$ y los nodos SC4 y SC5 la del conjunto $S = \{\{BF2, b\}\}$. Con esta información disponible, cada nodo *scan*, por cada tupla procesada, aplica los *bit filters* correspondientes sobre el atributo por el cual fueron creados. Sólo que uno de los *bit filters* retorne 0, entonces, la tupla puede ser descartada. Suponiendo que sólo utilizamos una función de hash h para los *bit filters*, el código a añadir al procesar una tupla sería:

```
for each s ∈ S do
    atr = obtener_atributo(tupla, s.atributo);
    valor_de_hash = h(atr);
    if s.bit_filter(valor_de_hash) == 0
    {
        descartar tupla;
        procesar siguiente tupla;
    }
end foreach;
```

La implementación de la ejecución de la técnica no es compleja, y es de importancia darse cuenta del bajo coste que requiere:

- Ocupación de memoria: sólo es necesaria la estructura S donde guardar la información, para cada nodo *scan*, acerca de los *bit filters* que puedan consultar.
- Consumo de cpu: Si consultamos los *bit filters* en los nodos *scan*, ya no hará falta hacerlo en la fase de *probe* del nodo hash join. Así, PDBF no implica replicación de trabajo alguna.

4.3 Configuración de la evaluación

Se ha desarrollado toda la evaluación sobre un un P-III a 550 MHz, con 512 MBytes de memoria RAM y un disco duro de 20GB IDE. El sistema operativo instalado es Linux Red Hat 6.2. Para la ejecución de nuestras pruebas la máquina se ha reservado de forma exclusiva, y por lo tanto, el sistema no está siendo compartido con ninguna otra aplicación.

PostgreSQL es el Sistema Gestor de Bases de Datos utilizado, completando tres codificaciones distintas del motor de PostgreSQL:

1. Motor original de PostgreSQL, sin *bit filters*.
2. Motor de PostgreSQL con *bit filters*.
3. Motor PostgreSQL con *bit filters* y la técnica PDBF.

La implentación realizada sobre el motor de PostgreSQL, no incluye ninguna modificación en el optimizador. De forma que, sólo se han realizado las modificaciones necesarias sobre PostgreSQL para ejecutar la consulta que se ha diseñado para nuestras pruebas.

PostgreSQL decide el número de particiones en el *Hybrid Hash Join* durante el tiempo de optimización de una consulta. Con la aplicación de nuestra estrategia, algunos nodos hash join acaban teniendo un número menor de tuplas provenientes de la relación de *build* . Esto implica que el número de particiones a realizar podría ser menor que el estimado a *priori* por el optimizador de PostgreSQL. Sin embargo, nuestras implementaciones mantienen el número de particiones estimado por el optimizador a coste de hacer nuestra estrategia menos eficiente.

En PostgreSQL el área de memoria asignada a cada nodo hash join se denomina *Sortmem*. Para el análisis de la técnica se han realizado ejecuciones con diferentes tamaños de *Sortmem*, desde tamaños muy restrictivos hasta tamaños que permiten la ejecución del nodo hash join en memoria. Los tamaños de *Sortmem* utilizados para cada una de las configuraciones varían de entre 5 MB (restrictivo) a 60 MB (hash join en memoria). Los *bit filters*, siempre se utilizan, a no ser que se sepa que su eficiencia va a ser nula, en cuyo caso, no son creados. Los *bit filters* se almacenan en una zona de memoria independiente del *Sortmem*.

Entorno de ejecución

Hemos generado una base de datos TPC-H [2] con un factor de escala 1 (1GB). Para la evaluación se ha aplicado la técnica a una consulta con siete configuraciones distintas sobre la base de datos proporcionada por el *benchmark* TPC-H.

```
select
p_name, p_partkey, sum(l_quantity)
from
    part,
    lineitem,
    partsupp,
    orders
where
    ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
```

and o_orderdate = ’x’
and ps_availqty> ’y’
and p_name like ’z’
group by
p_name, p_partkey

Nota: Los diferentes valores que asignemos a *x,y,* y *z* determinarán las restricciones de las tablas *orders*, *partsupp*, y *part* respectivamente.

La consulta es de tipo join estrella, relacionando la tabla de hechos *lineitem*, con las tablas de dimensión *part*, *partsupp*, y *orders*. Esto no limita, ni mucho menos, nuestra técnica a este tipo de consultas, sin embargo, es un buen ejemplo para la evaluación de los PDBF.

La Figura 4.3 muestra el plan de ejecución correspondiente a la consulta, además del uso de la técnica PDBF (lineas discontinuas). Podemos ver como el *scan* sobre $lineitem$ consulta los *bit filters* de HJ1, HJ2, y HJ3, mientras quel el *scan* sobre part sólo consulta HJ2.

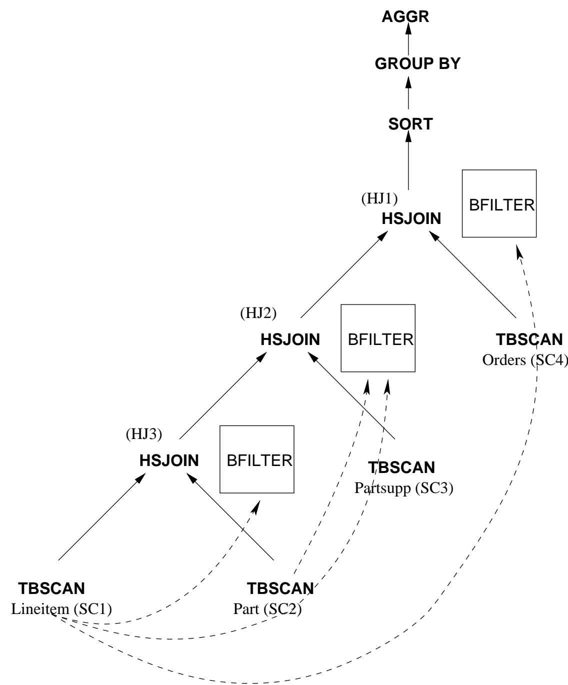


Figura 4.3 Consulta utilizada y su plan de ejecución.

Hemos usado las restricciones "x","y", y "z" sobre las tablas *part* (P), *partsupp*(PS), y *orders* (O), para controlar la cantidad de tuplas proyectadas por la relación de *build* de cada nodo hash join. De esta forma podemos variar las restricciones con el fin de obtener diferentes comportamientos de la misma consulta. En la Tabla 4.1 se muestran siete configuraciones distintas. La diferencia entre cada configuración es la selectividad de los nodos *scan* sobre las tablas part, partsupp y orders, que varía según las variables mencionadas. La misma Tabla, a través de la eficiencia de los *bit filters*, muestra la fracción de tuplas eliminadas por cada *bit filter* en los nodos hash join. La restricción sobre las relaciones de *build* de cada nodo hash join, modifica el número de valores distintos proyectados, repercutiendo, a su vez, en la selectividad del *bit filter* S_{bf}, y por lo tanto en su

eficiencia ($E_{bf} = 1 - S_{bf}$). Los datos de esta Tabla han sido extraídos de la información obtenida tras la ejecución de la consulta. Cada configuración puede ser vista como una consulta distinta y nos permite analizar el comportamiento de los PDBF bajo un amplio abanico de posibilidades.

Configuración	**Selectividad**			**Eficiencia *bit filter***		
#	**part**	**partsupp**	**orders**	**HJ3**	**HJ2**	**HJ1**
1	1.0	0.3	0.2	0.0	0.16	0.81
2	1.0	0.85	0.4	0.0	0.0	0.54
3	1.0	0.3	0.85	0.0	0.16	0.18
4	0.2	0.2	0.2	0.81	0.44	0.72
5	0.2	0.85	0.85	0.81	0.0	0.18
6	1.0	0.85	0.85	0.0	0.0	0.18
7	1.0	0.5	0.4	0.0	0.8	0.8

Tabla 4.1 Configuraciones realizadas.

4.4 Resultados

En esta Sección presentamos diferentes resultados obtenidos a través de las ejecuciones reales. Primero, evaluamos la consulta bajo las siete configuraciones mostradas en la Tabla 4.1 con el fin de mostrar las prestaciones de PDBF. Entonces, damos medidas para el volumen de E/S usando una única configuración base.

Tráfico de datos entre los nodos del plan y tiempo de ejecución

Las Figuras 4.4 y 4.5 muestran resultados del tráfico de datos (en millones de tuplas) y tiempo de ejecución. Las gráficas muestran resultados para la implementación original de PostgreSQL, para el uso de *bit filters* en los nodos en los que fueron creados, y para PDBF. Las gráficas referentes al tiempo de ejecución de la consulta, muestran resultados para diferentes tamaños de *Sortmem*, mientras que las gráficas referentes a tráfico de datos, muestran el tráfico total por cada nodo en el plan de ejecución (el tráfico de datos es independiente del tamaño de *Sortmem*). El tráfico de un nodo es equivalente a la cardinalidad de salida de ese nodo. Significar que sólo el tráfico de los nodos HJ3, HJ2, SC1, y SC2 (ver figura 4.3) se ven afectados por PDBF.

Mientras el tiempo de ejecución está directamente relacionado con la implementación de PostgreSQL, el tráfico ahorrado por PDBF sólo depende de los datos y puede ser considerado como un resultado genérico para cualquier SGBD.

Dividimos el análisis en dos grupos de ejecuciones:

- **Configuraciones #1, #2 ,#3 y #7** : Son las ejecuciones en las que hemos obtenido una significativa mejora tanto en el tiempo de ejecución como en el tráfico de datos entre nodos del plan. Como se muestra en las gráficas de estas ejecuciones, la mayor mejora en cuanto a tráfico de datos viene dada por los nodos HJ2, HJ3 y SC1. Esto es debido a que los *bit filters* creados en HJ1 a partir de la relación *orders*, son altamente eficientes para las configuraciones #1, #2 , y #7 (ver Tabla 4.1). A través de este bit filter, el nodo SC1 filtra gran parte del las tuplas leídas durante la lectura de la tabla *lineitem*, reduciendo así el volumen de datos a procesar por el resto de nodos del plan. Como se puede observar en las

gráficas, cuanto mayor es la disminución del tráfico de tuplas, mayor es la disminución en tiempo.

Otro aspecto importante es ver el efecto que tiene el tamaño del *Sortmem* sobre los tiempos de ejecución. Mientras que el tamaño del *Sortmem* apenas tiene efecto sobre la configuración #1, sí muestra una influencia significatva sobre las configuraciones #2, #3 y #7. Esto es debido a que el *bit filter* de HJ1 tiene una alta efectvidad (0.81) en la configuración #1, lo cual reduce los datos a procesar de la tabla *lineitem* hasta el punto que minimiza la E/S inclusive para ejecuciones que utilizan tamaños muy pequeños de *Sortmem*.

El último aspecto a resaltar es que para tamaños de *Sortmem* iguales a 60MB, las mejoras en tiempo de ejecución derivan de la reducción de datos a procesar por la cpu. En las tres implementaciones de PostgreSQL, un incremento en el tamaño de *Sortmem* reduce de forma considerable el total de E/S realizada. Así pues, es importante resaltar, que incluso en esos casos, el tiempo de ejecución ahorrado es significativo, dando más fuerza a nuestra estrategia.

- **Configuraciones #4, #5 y #6**: Estas ejecuciones, aunque presentan buenos resultados, no alcanzan a obtener las mejoras obtenidas por el resto. Los nodos *scan* de las configuraciones #4 y #5 tienen una selectividad muy baja sobre la tabla *part*. Esto supone que la relación de *build* de la operación HJ3 cabe en memoria, reduciendo la E/S de esta operación al mínimo obligatorio, siendo de escaso coste y con una cardinalidad de salida muy pequeña en cualquiera de los tres motores de PostgreSQL testeados. Además, casi todas las tuplas descartadas por la técnica en SC1 (el *scan* sobre *lineitem*), se debe a la acción del *bit filter* de HJ3, ahorrándonos sólo el trabajo que supone proyectar estas tuplas del nodo SC1 a HJ3.

  En la configuración #4 se obtienen resultados algo mejores, ya que la selectividad sobre la relación *partsupp* y *orders* es más baja, mejorando así la efectividad de los *bit filters* de HJ1 y HJ2 respecto a la configuración #5.

  La configuración #6 no tiene restricción sobre *part*, y una selectividad muy alta sobre *partsupp* y *orders*, repercutiendo en una mala eficiencia de los bit filter, y en consecuencia un uso deficiente de la técnica.

  Significar que, en estos casos, el optimizador optaría por el no uso de los *bit filters* debido a su escasa efectividad comparado con el tamaño que ocupan. Sin embargo, el escaso beneficio obtenido en estos casos pone de manifiesto que la técnica PDBF, no supone sobrecarga alguna incluso en situaciones en las que la mejora a obtener es mínima.

A modo de resumen, podemos decir que los mejores resultados en las configuraciones donde es posible el uso de PDBF se obtienen cuando los *bit filters* utilizados desde los nodos hoja del plan de ejecución, son los creados en los nodos superiores del plan. Cuando esto ocurre, el tráfico de datos entre nodos se reduce de forma significativa, y en consecuencia el tiempo de ejecución.

E/S para la configuración #1

La Figura 4.4 muestra que, para la configuración #1, PDBF mejora el tiempo de ejecución en más de un 50% respecto la implementación original de PostgreSQL y la básica con los *bit filters*. En este apartado, analizamos las implicaciones de la E/S en el tiempo de ejecución.

La Figura 4.6 muestra la E/S realizada por cada uno de los tres nodos hash join en la configuración #1. No incluimos la E/S generada por los nodos *scan* porque no sufre modificación alguna. Como

se ha explicado en el Capítulo 2 (Sección 2.4), la cantidad de E/S en las operaciones de hash join varía dependiendo de la cantidad de datos de la relación de *build* y la cantidad de memoria disponible por el nodo hash join. Como se muestra en 4.6, si la cantidad de datos de la relación de *build* cabe en memoria, entonces la cantidad de E/S realizada por los nodos hash join es cero.

Las gráficas muestran que PDBF mejora en más de un 50% con respecto a las otras implementaciones. En el nodo HJ1, no hay mejora de PDBF respecto al uso de *bit filters*. Esto es debido a que HJ1 es el último join del plan de ejecución, y el *bit filter* creado ya reduce la relación de *probe* antes de ser particionada. La cantidad de E/S en HJ2 y HJ3 se reduce debido a que los *bit filters* son aplicados desde los nodos hoja del plan de ejecución. Nótese que a medida que se incrementa el tamaño del *Sortmem*, la E/S de la implementación original y de los *bit filters*, se reduce en menos proporción que el de la E/S de PDBF.

Como se muestra en 4.6 el uso de PDBF con un *Sortmem* de 60MB se reduce la cantidad de E/S a las operaciones de lectura obligatorias, así como para las otras dos implementaciones de PostgreSQL. Sin embargo, nuestra estrategia muestra una reducción significativa en el tiempo de ejecución (ver Figura 4.4). Esto es debido a que la cantidad de trabajo de cpu se ve reducido gracias al decremento de tráfico de datos a procesar.

4.5 Conclusiones

En este Capítulo hemos propuesto *Pushed Down Bit Filters* (PDBF), una nueva estrategia que consiste en utilizar los *bit filters* creados en los nodos join del plan de ejecución, desde los nodos hoja del plan.

Podemos recalcar las características más importantes de nuestra técnica de la siguiente forma:

1. PDBF disminuye, de forma muy significativa, el tráfico de datos entre los nodos del plan de ejecución. Este hecho repercute de forma directa en la cantidad de trabajo a realizar por los nodos intermedios del plan de ejecución y, en consecuencia, sobre el tiempo de cpu.

2. La reducción del tráfico de datos conduce a una reducción de la E/S en algunas consultas, viéndose reflejada en una clara mejora en el tiempo de ejecución de las consultas. El beneficio obtenido puede variar dependiendo de si las mejoras obtenidas son respecto a tráfico de datos, respecto a la E/S, o ambos a la vez. Para nuestras ejecuciones, en el mejor de los casos se ha llegado a obtener una reducción en el tiempo de ejecución de más de un 50%.

3. El mecanismo deejecución de PDBF es relativamente senzillo, tan sólo requiere el uso de los *bit filter* ya generados por los nodos de join del plan de ejecución. Además, la técnica es fácil de implementar y requiere de simples estructuras de datos a manejar por los nodos hoja del plan.

En definitiva, los claros beneficios obtenidos por PDBF durante la ejecución de una consulta, unido al bajo coste de su implementación, hacen de nuestra propuesta una técnica a sugerir para los SGBDs paralelos de hoy dia.

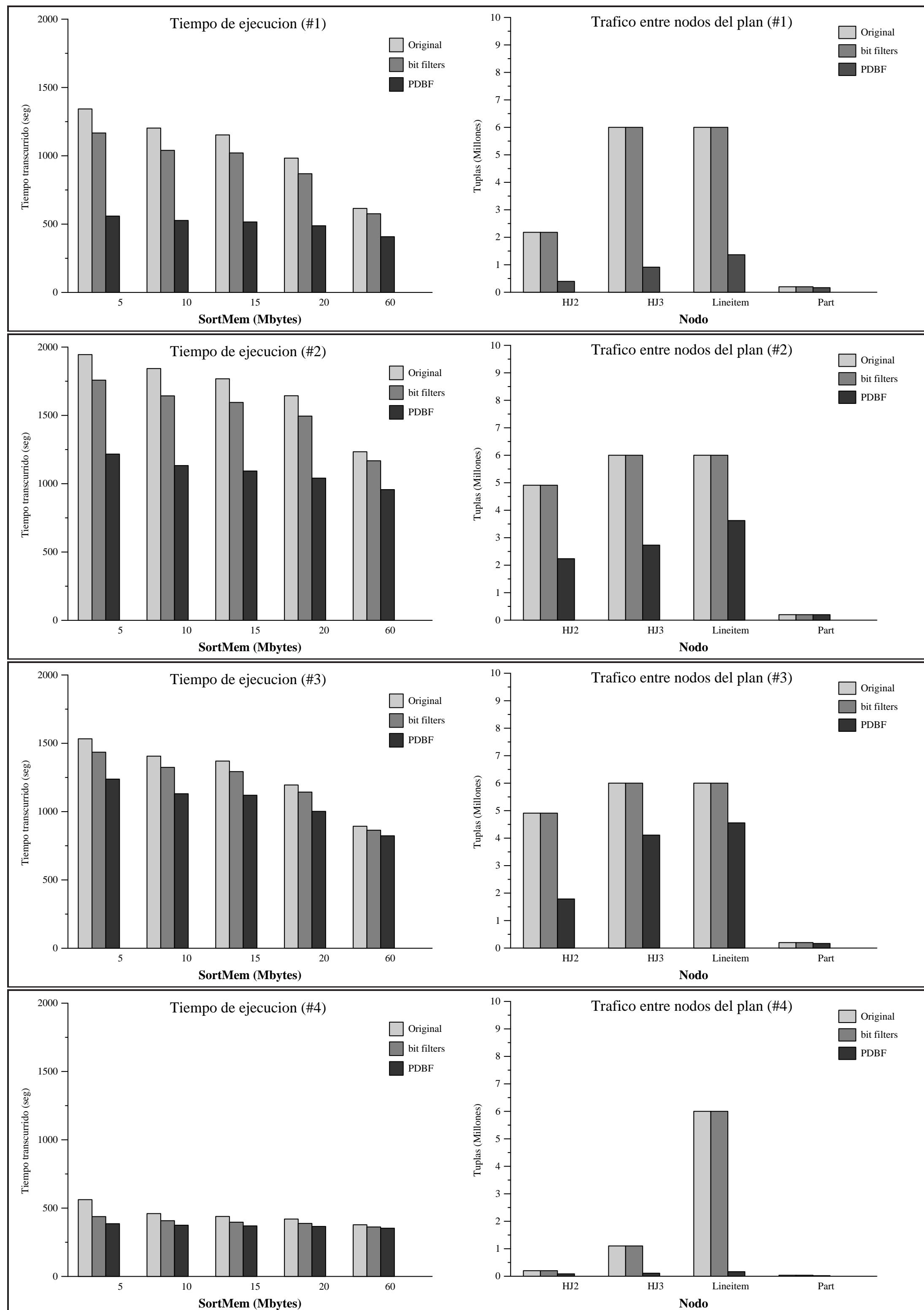


Figura 4.4 Tiempo de ejecución y tráfico de datos entre nodos para las configuraciones #1, #2, #3 y #4.

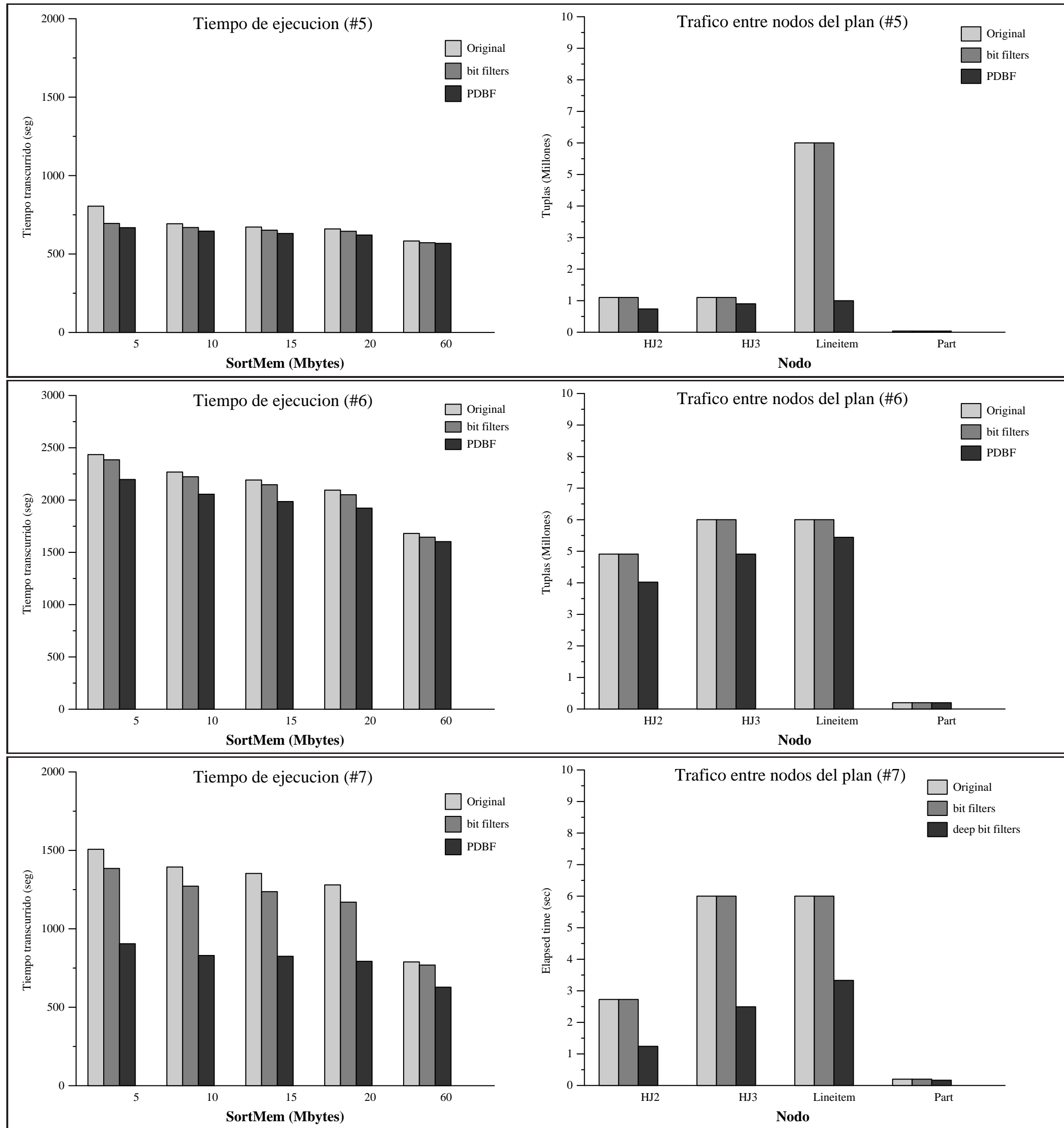


Figura 4.5 Tiempo de ejecución y tráfico de datos entre nodos ara las configuraciones #5, #6 y #7.

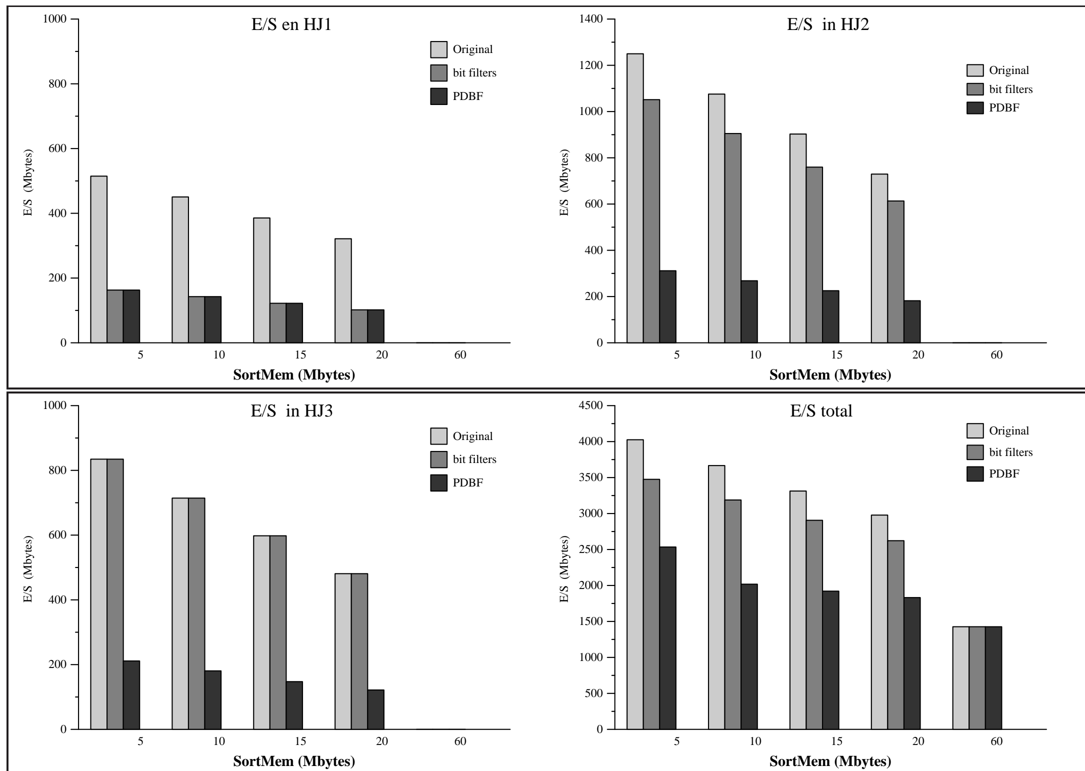


Figura 4.6 E/S para los nodos hash join.

5

STAR HASH JOIN

5.1 Introducción

En los dos anteriores Capítulos hemos presentado dos técnicas distintas. Por un lado la técnica *Remote Bit Filters* (RBF), que se aplica a arquitecturas sin recursos compartidos y tiene como objetivo reducir la comunicación de datos en sistemas distribuidos. Por otro lado la técnica *Pushed Down Bit Filters* (PDBF), que es aplicable a arquitecturas paralelas con recursos compartidos, y su objetivo reside en reducir el procesado de datos intra-operacional en el plan de ejecución de una consulta.

Este Capítulo centra su estudio en la ejecución de consultas ad hoc de tipo join estrella en arquitecturas *cluster*, y proponemos un nueva técnica, el *Star Hash Join* (SHJ) que fusiona RBF y PDBF con el fin de reducir el volumen de datos a procesar y comunicar de la tabla de hechos durante la ejecución de este tipo de consultas en arquitecturas *cluster*.

5.1.1 Trabajo relacionado y motivación

Aunque los sistemas *cluster* alivian el problema de la E/S y el procesado masivo de datos, el desarrollo de técnicas para acelerar la ejecución de consultas de tipo join estrella son un foco muy importante de investigación. Durante el procesado de un join estrella en arquitecturas *cluster*, la cantidad de datos a extraer, procesar, y comunicar de la tabla de hechos sigue suponiendo un serio cuello de botella. Técnicas como el *bitmap join* [64], *Multi Hierarchical Clustering* [31; 57; 62], o vistas materializadas [76] tratan de acelerar la ejecución de este tipo de consultas.

El uso de vistas materializadas en consultas ad hoc es muy limitado debido a que el término ad hoc indica que no sabemos cuales serán los patrones de conducta que va a tener la consulta. El *bitmap join* (BJ), introducido por O'Neil y Graefe es la base de muchos otros proyectos de investigación y está cimentado en el uso de índices *bitmap* [22; 83]. BJ presenta, de forma teórica, la ejecución ideal del join estrella, que se ejecuta mediante rápidas operaciones AND/OR de mapas de bits en memoria, y reduce al mínimo los datos a procesar de la tabla de hechos. El algoritmo, detallado en el Capítulo 2 (Sección 2.2), tiene como principal inconveniente el uso masivo de índices, y un requerimiento muy elevado de memoria para cargar los mapas de bits, que pueden alcanzar tamaños mayores que el tamaño total de la base de datos.

La alternativa al uso de índices es el *Multi Hierarchical Clustering* (MHC). MHC, también detallado en el Capítulo 2 (Sección 2.2), realiza un *clustering* físico de datos con el objetivo de

limitar la E/S generada sobre la tabla de hechos en consultas de tipo join estrella. En este caso, la tabla de hechos se crea especificando una o más claves de las tablas de dimensión, que son las utilizadas para organizar de forma jerárquica la tabla de hechos. De este modo, la tabla de hechos está jerárquicamente organizada a través de múltiples dimensiones, y las consultas de tipo join estrella se convierten en consultas por rango sobre la tabla de hechos, reduciendo la E/S realizada sobre ésta. El principal inconveniente de MHC reside en el hecho que las consultas tienen que estar restringidas por los mismos atributos por los que se ha multidimensionado la tabla de hechos, de otro modo, no son efectivas. En el caso de consultas ad hoc la probabilidad de que esto ocurra son bajas.

En este Capítulo presentamos la técnica *Star Hash Join* (SHJ) con el fin de acelerar la ejecución de consultas ad hoc de tipo join estrella en arquitecturas *cluster*, y en especial aquellas cuyos nodos tienen una configuración SMP, también conocidas como sistemas CLUMP [29]. Los principales objetivos de SHJ son:

- Reducir la comunicación de datos entre nodos, y la E/S causada por los operadores de join del plan de ejecución.

- Ejecución ad hoc de la técnica. Es decir, que la ejecución de la técnica no esté sujeta a ningún tipo de suposición acerca de la naturaleza de la consulta.

- Evitar el uso de estructuras auxiliares que, como los índices, requieran de costes de manutención y recursos de memoria.

5.1.2 Contribuciones

Podemos enumerar las contribuciones del trabajo presentado en este Capítulo como:

1. La propuesta de la técnica *Star Hash Join* (SHJ), que explota el uso de los bit filters, y consiste en una generalización de los *Pushed Down Bit Filters* (PDBF) a arquitecturas CLUMP a través del uso de *Remote Bit Filters* (RBF).

2. La construcción de un modelo analítico para la evaluación y comparación de BJ, MHC y SHJ.

3. El análisis y comparación de las técnicas SHJ, BJ y MHC, junto con RBF y PDBF. El análisis demuestra que, aunque cada una de estas técnicas se centra bien en el problema de la E/S para arquitecturas SMP, o bien en el problema de la comunicación de datos para arquitecturas *cluster*, resulta difícil encontrar una técnica que de un beneficio en todos los casos.

4. A partir del análisis realizado, la propuesta de una solución híbrida entre SHJ y MHC, que reduce tanto la comunicación entre nodos, como la E/S dentro de cada nodo en un mayor número de situaciones que las técnicas previamente propuestas.

5.2 Star Hash Join (SHJ)

En esta Sección explicamos el *Star Hash Join* (SHJ). Primero introduciremos el contexto de ejecución del algoritmo, y posteriormente procederomos a la explicación del mismo.

5.2.1 Contexto de ejecución. Esquema de tipo Star Hash Join

Entendemos por esquema de tipo *Star Hash Join* el perfil de plan de ejecución que se necesita para lleva a cabo nuestra técnica.

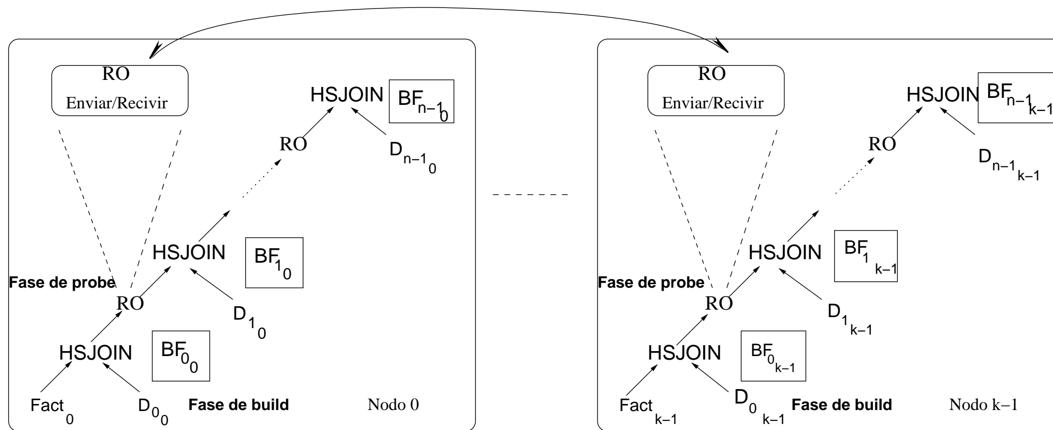


Figura 5.1 Esquema *Star Hash Join* para un *cluster* con k nodos.

Asumiendo un particionado hash de la base de datos a través de los nodos del sistema, una consulta de tipo join estrella con n tablas de dimensión y la tabla de hechos, puede ser ejecutada mediante n operadores de hash join implementados a través del algoritmo Hybrid Hash Join, y concatenados a través de la fase de probe tomando una forma de *left-deep tree* [47]. La Figura 5.1 muestra un ejemplo de esquema *Star Hash Join* para un arquitectura formada por k nodos. Durante el desarrollo de este Capítulo utilizamos las siguientes definiciones:

- D_{i_j} es la porción de la i^{esima} tabla de dimensión almacenada en el nodo j, y BF_{i_j} es el *bit filter* creado durante la ejecución de su fase de build.

- $Fact_j$ es la porción de la tabla de hechos (*Fact table*) almacenada en el nodo j.

- RO es el operador de reparticionado responable de enviar, de forma selectiva, los datos al resto de nodos del sistema, así como de recibir datos de cada uno de esos nodos.

Tal y como se explicó en el Capítulo 2, en un esquema estrella es frecuente que las tablas de dimensión estén particionadas a través de sus respectivas claves primarias, y la tabla de hechos esté particionada por una de sus claves foráneas. De este modo, sólo uno de los join entre una tabla dimensión y la tabla de hechos puede ser colocalizado. El resto de joins son no colocalizados, y necesitan de la presencia de un operador de reparticionado (RO) durante la fase de probe del plan de ejecución.

Notar que podría darse el caso de que, por cuestiones de optimización, algún nodo hash join hiciera un *broadcast* de la fase de build en lugar de un reparticionado selectivo de la fase de probe. Este caso no es contemplado en nuestro ejemplo a lo largo del Capítulo por cuestiones de simplicidad. Nótese sin embargo, que cualquier variación en el plan de ejecución de la Figura 5.1 es posible, y nuestra técnica seguiría funcionando con normalidad.

5.2.2 Generalización de PDBF a arquitecturas cluster

SHJ generaliza el uso de *Pushed Down Bit Filters* (PDBF) con el fin de acelerar la ejecución de consultas de tipo join estrella en arquitecturas CLUMP. El algoritmo SHJ utiliza la técnica de los *Remote Bit Filters* (RBF) para realizar dicha generalización.

Para una mejor comprensión, SHJ utiliza la variante *Remote Bit Filters Broadcasted* (RBF$_B$), aunque *Remote Bit Filters with Requests* (RBF$_R$) también se podría aplicar, con las diferencias entre ambas técnicas explicadas y detalladas en el Capítulo 4. RBF$_B$ realiza el *broadcast* de todos los *bit filters* creados localmente durante la fase de build a todos lo nodos involucrados en el proceso del hash join no colocalizado. De esta forma, antes de comunicar una tupla durante la fase de probe, la tupla es chequeada contra el *bit filter* del nodo destino, y a ser posible, es descartada antes de ser enviada. La Figura 5.2-(a) muestra la aplicación de RBF$_B$ bajo el esquema *Star Hash Join* explicado anteriormente. Para cierto nodo j, un nodo hash join entre una tabla de dimensión (D_{i_j}) y la tabla de hechos ($Fact_j$), mantendrá los *bit filters* $BF_{0_j}, BF_{1_j}, ..BF_{n-1_j}$ en su memoria local, uno por cada una de las n tablas de dimensión.

Los PDBF, como se vió en el Capítulo anterior, están destinados a ahorrar tráfico de datos intra-operacional en cada nodo. PDBF utiliza los *bit filters* generados en los nodos superiores para filtrar tuplas en los nodos hojas del plan de ejecución. Esto ahorra tanto el procesamiento de datos que se sabe de antemano que no forman parte del resultado final, como E/S de nodos intermedios del plan de ejecución. La Figura 5.2-(b) muestra la aplicación de PDBF bajo el esquema *Star Hash Join*. Cada tupla leída de la tabla de hechos es chequeada contra los n bit filters. Si la entrada asociada a la tupla en proceso, para alguno de los bit filters, es igual a 0, entonces la tupla se puede descartar. Nótese que PDBF no puede ser utilizado en arquitecturas *cluster* porque no es posible, usando sólo los *bit filter* locales, filtrar datos que podrían ser transmitidos a nodos remotos.

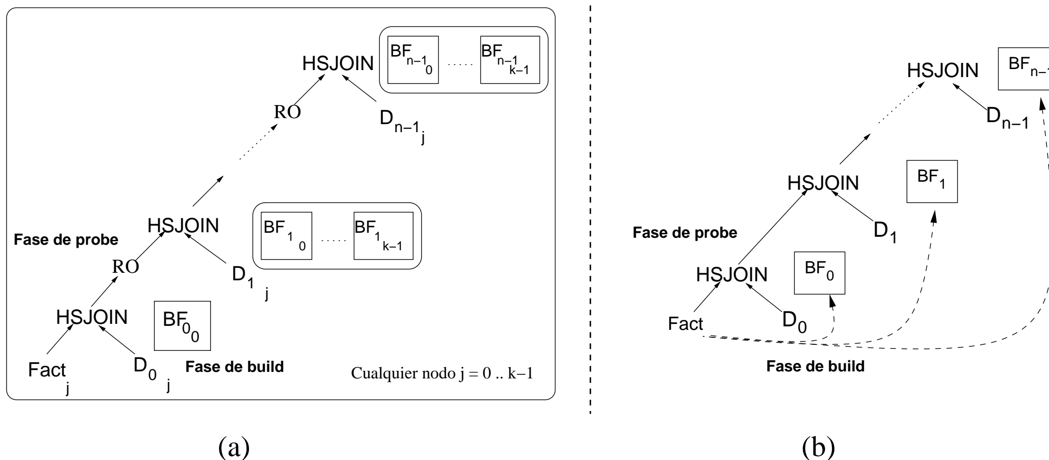
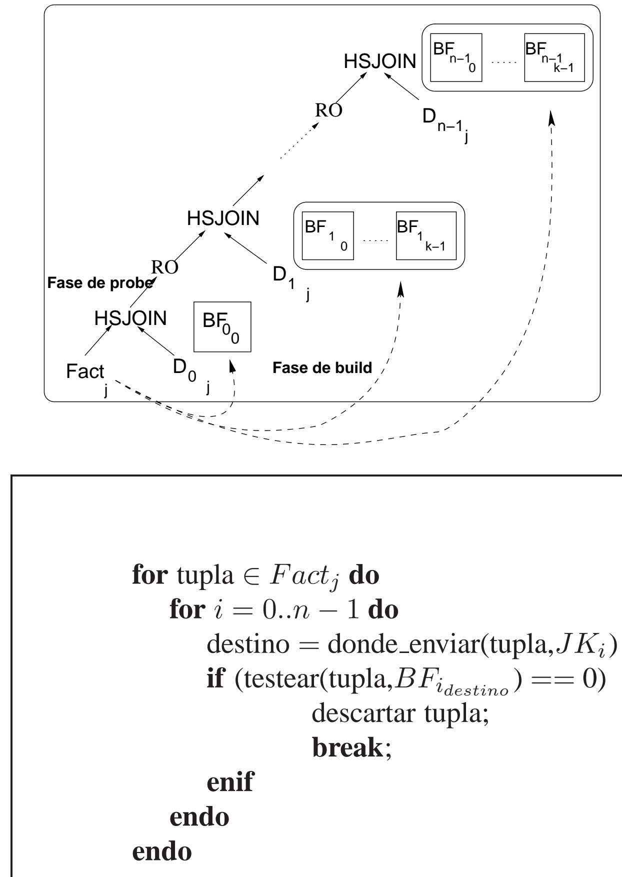


Figura 5.2 (a) *Remote Bit Filters Broadcasted* (RBF$_B$); (b) *Pushed Down Bit Filters* (PDBF).

5.2.3 El algoritmo

La estrategia seguida por SHJ se explica gráficamente en la Figura 5.3. RBF$_B$ se aplica para mantener todos los *bit filters* del join estrella en cada nodo. Por otro lado, PDBF permite a los nodos hoja del plan de ejecución acceder a todos los *bit filters* de la consulta. De este modo, cada

tupla de la tabla de hechos (*Fact*), es chequeada contra los *bit filter* del nodo destino, y una tupla se transmite sólo si tenemos la certeza de que tiene posibilidad de hacer join con alguna tupla del nodo destino. Así pues, SHJ extiende el uso de los PDBF a arquitecturas *cluster*. Además, obtiene un mayor beneficio en términos de comunicación de datos que el uso de RBF_B a solas.



for tupla $\in Fact_j$ **do**
    for $i = 0..n-1$ **do**
        destino = donde_enviar(tupla,JK_i)
        if (testear(tupla,$BF_{i_{destino}}$) == 0)
            descartar tupla;
            break;
        enif
    endo
endo

Figura 5.3 El algoritmo Star Hash Join.

En la misma Figura 5.3 mostramos la versión algorítmica de SHJ. Dado un nodo j y n joins entre las tablas de dimensión (D_{i_j}, $i = 0..n-1$), y la tabla de hechos ($Fact_j$), denominamos a las claves de join como JK_i. Así pues, cada tupla leída de la tabla de hechos $Fact_j$ es procesada de la siguiente forma:

1. Para cada join con las tablas de dimensión, aplicamos la función de reparticionado utilizada para distribuir los datos entre nodos, sobre la clave de join: donde_enviar(JK_i). De esta foma, averiguamos el nodo destino de la tupla en proceso. Las claves de join en una consulta de join estrella son normalmente las claves primarias, las cuales son, a su vez, las claves de particionado en las tablas de dimensión. De modo que, la clave de join JK_i almacenada en cada tupla de la tabla de hechos, es la única información necesaria para determinar el nodo destino de una tupla.

2. Entonces, la tupla en proceso es chequeada contra los *bit filters* del nodo destino: testear(tupla, $BF_{i_{destino}}$). Si la función retorna cero, entonces la tupla es filtrada, de otra forma la tupla tiene que ser procesada.

Así pues, una tupla de la tabla de hechos sólo es procesada si:

$$\forall i : i = 0..n - 1 : testear(BF_{i_{destino}}) = 1$$

de este modo, el algoritmo SHJ elimina gran parte del procesado de datos de la tabla de hechos desde las hojas del plan de ejecución, reduciendo comunicación de datos entre nodos, y procesado de datos dentro de cada nodo, evitando consumo de cpu y E/S inncesaria.

5.3 Configuración de la evaluación

Realizamos nuestra evaluación analizando un entorno similar al usado por el *benchmark* TPC-H [2]. Analizamos la ejecución de una consulta basada en el *benchmark* TPC-H sobre una arquitectura con 5 nodos con una configuración SMP, cada nodo con 32 procesos compartiendo un total de 256GB de memoria principal y 512 discos. Una configuración similar a la utilizada por IBM en un *benchmark* TPC-H de 10TB.

Entorno de ejecución

Asumimos una base de datos TPC-H de 10TB, particionada entre nodos utilizando un particionado hash. Para nuestro análisis, hemos usado una consulta basada en la consulta 9 de TPC-H:

```
select
nation, o_year, sum(amount) as sum_profit from
(
select
    n_name as nation,
    year(o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount) - ps_supplycost*
    l_quantity as amount
from
    tpcd.part,tpcd.supplier,
    tpcd.lineitem, tpcd.partsupp,
    tpcd.orders, tpcd.nation
where
    s_suppkey = l_suppkey
    and ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
    and s_nationkey = n_nationkey
    and p_name like x
    and n_nationkey > y
    and o_orderpriority = z
    and ps_availqty > w
    ) as profit
group by
    nation, o_year
order by
    nation,
```

o_year desc;

Nota: Los diferentes valores que asignemos a *x,y,z, y w* determinarán las restricciones de las tablas de dimensión.

Esta consulta, es un modelo de consulta join estrella, donde se realiza el join entre la tabla de hechos, $lineitem$, y las 4 tablas de dimensión $partsupp$, $part$, $supplier$ y $orders$. También se realiza el join entre las tablas de dimensión $supplier$ y la tabla $nation$. La consulta original sólo presenta una restricción sobre $part$, de modo que la hemos modificado añadiendo tres condiciones sobre $nation$, $orders$ y *partsupp* de cara a restringir la consulta a un conjunto de datos más pequeño. En la Figura 5.4, mostramos la parte del plan de ejecución de la consulta que es de nuestro interés. Como se puede apreciar tiene un perfil de esquema *Star Hash Join* explicado en este Capítulo. Siguiendo el ejemplo mostrado en el Capítulo 2, las tablas de dimensión han sido particionadas por su clave primaria, mientras que $lineitem$ a través de la clave foránea $l_orderkey$. De este modo, sólo el join entre $lineitem$ y $orders$ puede ser colocalizado, el resto necesitan de reparticionado de datos para su ejecución. La memoria disponible para cada hash join es de 1.3GB, y el grado de paralelismo SMP es de 32, es decir, 32 procesos trabajarán en paralelo compartiendo recursos en cada nodo.

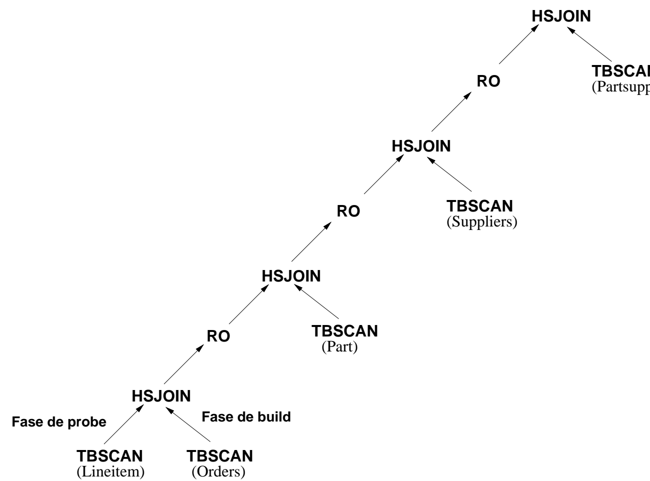


Figura 5.4 Plan de ejecución.

Grupos	S_{orders}	S_{part}	$S_{suppliers}$	$S_{partsupp}$
1	0.7	0.7	0.2	0.2
2	0.2	0.2	0.7	0.7
3	0.7	0.2	0.2	0.7
4	0.2	0.7	0.7	0.2
5	0.7	0.7	0.7	0.2
6	0.2	0.7	0.7	0.7

Tabla 5.1 Tabla de grupos de selectividades aplicadas a SHJ.

La selectividad de las tablas de dimensión varía dependiendo de los valores *x, y, z* y *w* que determinan las restricciones sobre las tablas de dimensión. En la Tabla 5.1 mostramos los distintos grupos de selectividades aplicados sobre las tablas de dimensión, y que utilizamos con el fin de analizar un ámplio abanico de posibles situaciones a ser comparadas. Los *bit filters* son utilizados en todos los nodos hash join, sin importar la selectividad de la relación de build. La fracción de falsos positivos [14] (P) para todo *bit filter* es $P = 0.05$.

Técnicas analizadas

En esta Sección realizamos un análisis detallado de las técnicas orientadas a la ejecución del join estrella y explicadas a lo largo de esta tesis: *bitmap join* (BJ), *Multi Hierarchical Clustering* (MHC), y *Star Hash Join* (SHJ). También incluimos en nuestro análisis la técnica *Remote Bit Filters* (RBF) con el fin de compararla con SHJ. Por otra parte PDBF no es una técnica que podamos comparar con el resto pues no está preparada para ser ejecutada en arquitecturas *cluster*, además, su uso ya está incluido en SHJ.

En el caso de MHC, analizamos su comportamiento dependiendo de los atributos utilizados para multidimensionar de forma jerárquica la tabla de hechos. En la Tabla 5.2 se especifican las diferentes configuraciones para MHC. Por ejemplo, de los atributos por los que se ha multidimensionado MHC-1D, sólo el atributo *o_orderpriority* aparece en la consulta especificada para nuestra evaluación. Para MHC-2D, coinciden los atributos *o_orderpriority* y *p_name*, y para MHC-4D coinciden los cuatro atributos por los que se acota el resultado de la consulta (notar que *s_nationkey = n_nationkey*).

MHC-dimensiones	o_orderpriority	p_name	s_nationkey	ps_availqty
MHC-1D	X	-	-	-
MHC-2D	X	X	-	-
MHC-4D	X	X	X	X

Tabla 5.2 Tabla de los atributos que aparecen en la consulta para nuestro análisis, y por los que la técnica MHC se ha dimensionado.

Modelo analítico

En el Apéndice B se detalla un modelo analítico que determina, bajo un modelo de plan de ejecución como el especificado en la Figura 5.4, la E/S y el consumo de cpu de las técnicas BJ, SHJ, y MHC. Para la técnica RBF$_B$ hemos empleado el modelo analítico para los *Remote Bit Filters* del Apéndice A.

5.4 Resultados

Todos los resultados presentados a continuación se basan en los modelos analíticos realizados para las técnicas evaluadas, bajo la configuración especificada en la Sección anterior.

Los resultados mostrados en las siguientes secciones aplican al beneficio obtenido por las técnicas BJ, MHC, RBF$_B$, y SHJ comparado con la ejecución base, sino se específica de otra forma. La ejecución base utiliza los *bit filters* de forma local y sin ninguna técnica aplicada. También, para los resultados que se van a mostrar, el eje de abcisas suele estar dividido por los 6 grupos mostrados en la Tabla 5.1.

Se recuerda que, como ya se explicó en el Capítulo 2, la técnica BJ es, desde un punto de vista teórico, la ejecución ideal de un join estrella, y reduce al mínimo los datos de la tabla de hechos a procesar. Por otro lado MHC y SHJ, son técnicas que intentan implementar el join estrella de una forma práctica, y obteniendo unos resultados lo más próximos a los ideales teóricos del BJ. Así pues, los mejores resultados serán los de la técnica que más se acerque a los resultados obtenidos por BJ.

Comunicación y E/S de los operadores de join

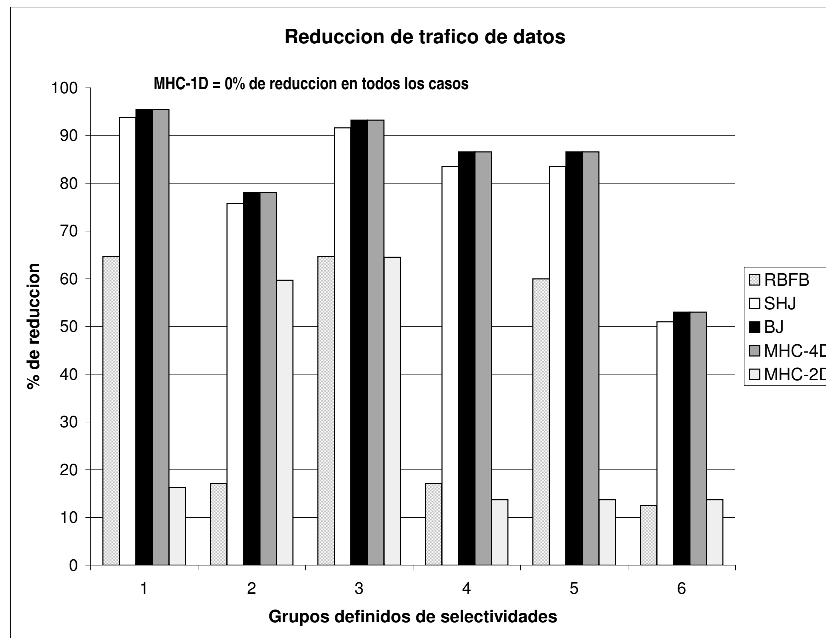
La Figura 5.9-(a) muestra el porcentaje de reducción de comunicación de datos entre nodos, mientras que la Figura 5.9-(b) muestra el porcentaje de E/S ahorrado por los operadores de join durante su ejecución. Las tendencias son similares en ambas gráficas:

- BJ y MHC-4D obtienen los mejores resultados, especialmente cuando la selectividad global de la consulta es baja, como en los grupos 1 y 3, en los que se observa un ahorro por encima del 90% en comunicación de datos, y por encima del 40% en E/S durante las operaciones de join. Los resultados para MHC-D4 son los óptimos, pues todos los atributos por los que se ha restringido la consulta coinciden con los que se ha multidimensionado la tabla de hechos.

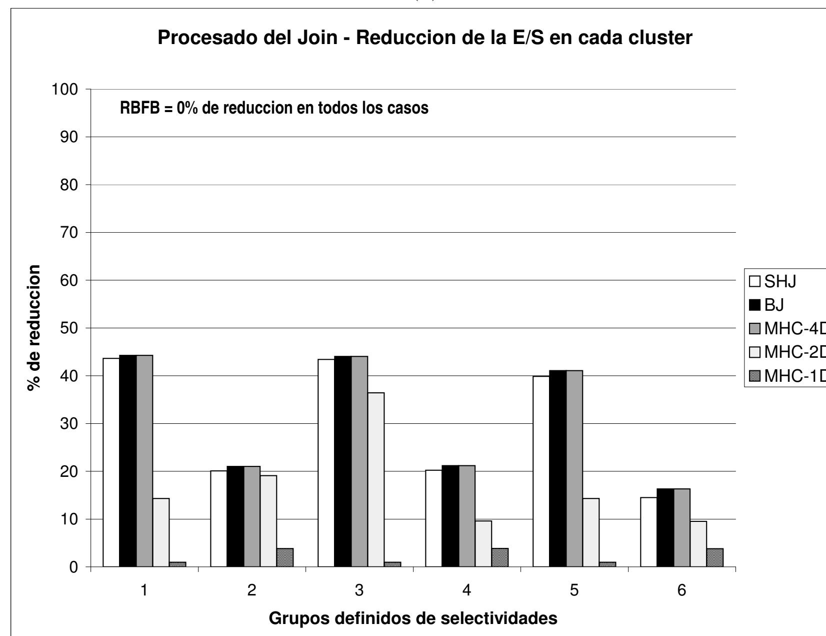
- SHJ alcanza unos resultados muy similares a los obtenidos por BJ, MHC-D4. La pequeña diferencia se explica a través de la fracción de falsos positivos P intrínseca en los *bit filters* utilizados por SHJ. También es importante observar que SHJ siempre reduce más comunicación de datos que RBF_B a solas: siempre por encima de un 25%, y en el mejor de los casos en un 65%.

- Cuando la tabla de hechos no ha sido multidimensionada a través de los atributos utilizados en la consulta para restringir las tablas de dimensión, entonces, MHC no obtiene buenos resultados. Por ejemplo, en el caso de MHC-1D, sólo coincide con el atributo *o_orderpriority* de la tabla *orders*, que está colocalizada. En este caso, el ahorro de comunicación de datos es nulo. También, MHC-2D, en los grupos 1, 4, y 5, obtiene pequeños porcentajes de beneficio tanto en E/S durante las operaciones de join, como en comunicación de datos entre nodos. Para estos grupos, las selectividades más bajas aplican a los atributos de las tablas de dimensión *suppliers* y *ps_partsupp*, los cuales no han sido utilizadas en MHC-2D para multidimensionar la tabla de hechos. Por lo tanto, para estos grupos, MHC-2D obtienen resultados pobres.

- SHJ obtiene los mejores beneficios cuando las selectividades bajas aplican a las tablas de dimensión no colocalizadas en los nodos superiores del plan de ejecución (grupos 1, 3, y 5). Esto era de esperar pues, en estos casos, al igual que pasaba con PDBF, si las selectividades bajas están en los nodos superiores, entonces, se filtran más datos que de otra forma deberían de ser procesados por los nodos inferiores del plan. Opuestamente, en el grupo 6 se observan los beneficios más bajos para cualquier técnica ya que la selectividad más baja aplica a la tabla de dimensión colocalizada, *orders*. En estos casos, en la misma ejecución base, la selectividad del primer join es muy baja, y la mayoría de los datos de la tabla de hechos se eliminan desde un buen principio, siendo escaso el beneficio a obtener por cualquera de las técnicas.



(a)



(b)

Figura 5.5 (a) Reducción de datos comunicados entre nodos; (b) Reducción de E/S realizada por los operadores de join.

E/S sobre la tabla de hechos

La Figura 5.6 muestra el ahorro de E/S durante la lectura de la tabla de hechos. SHJ y RBF_B actúan una vez se leen los datos de la tabla, así, estas dos técnicas no afectan a la E/S sobre la

tabla de hechos, que se ha de leer entera. Al igual que en la en los resultados anteriores, BJ y MHC-4D ahorran la máxima E/S posible. El hecho más significativo reside en que MHC-1D y MHC-2D, al contrario de lo observado en los resultados anteriores, esta vez sí obtienen buenos resultados, especialmente en los casos donde la selectividad es baja en las tablas de dimensión $orders$ y $suppliers$ (grupos 2, 3, y 4), que son las tablas que contienen los atributos utilizados por MHC-1D y MHC-2D para multidimensionar la tabla de hechos.

Los buenos resultados de MHC eran de esperar, pues cuando la tabla de hechos está físicamente dimensionada por algunos de los atributos que restingen las cuatro tablas de dimensión en el join estrella, entonces, el acceso a la tabla de hechos es directo a los bloques de datos dentro de esas dimensiones, evitando así E/S innecesaria.

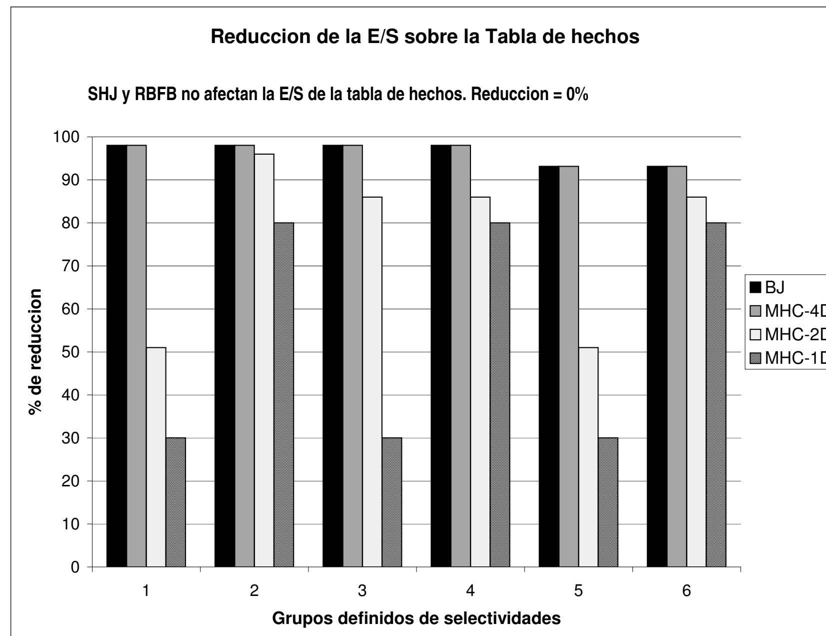
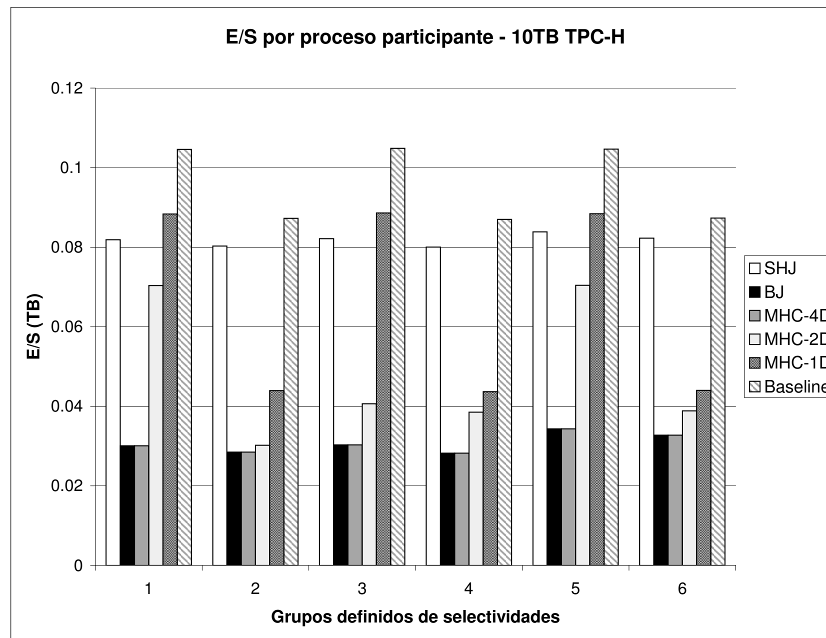


Figura 5.6 Ahorro de E/S en el *scan* sobre la tabla de hechos.

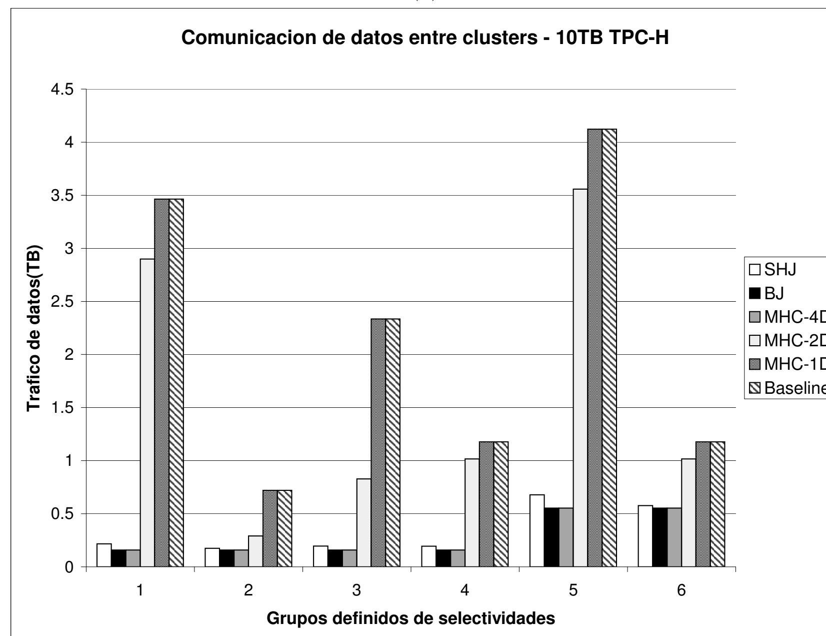
Communicación contra E/S

La Figura 5.7-(a) muestra la cantidad de comunicación de datos entre nodos durante la ejecución del join estrella. La Figura 5.7-(b) muestra los resultados en TB para la E/S. Para ambas gráficas no se muestran los resultados para RBF$_B$ ya que aportan poca información desde el momento en que es una técnica ya incluida en SHJ, y que ya ha quedado de manifiesto que SHJ siempre obtiene mejores resultados que RBF$_B$. Las conclusiones más importantes que podemos extraer de ambas gráficas son las siguientes:

- **Comunicación de datos**. SHJ es la técnica que, en un mayor número de casos, más se acerca a los resultados óptimos teóricos obtenidos por BJ. MHC sólo consigue buenos resultados cuando la tabla de hechos ha sido multidimensionada por los atributos que restringen las tablas de dimensión en el join estrella, MHC-4D. Así pues, SHJ es la técnica a escoger cuando la comunicación de datos es un problema.



(a)



(b)

Figura 5.7 (a) Comunicación de datos entre nodos; (b) E/S realizada.

- **E/S**. Los resultados relativos a E/S difieren de los obtenidos para la comunicación de datos. En un join estrella, el peso de la E/S reside en leer la tabla de hechos. De modo que, aunque SHJ, como se discutió a través de la Figura 5.9-(b), reduce de forma significativa la E/S de

los operadores de join, no ocurre lo mismo con la tabla de hechos, sobre la cual tiene un efecto nulo. Así pues, la mejor alternativa para reducir la E/S durante un join estrella sería el uso de MHC, que como se aprecia en la figura 5.6 reduce la E/S sobre la tabla de hechos en un amplio rango de casos.

- Un hecho observable es el mayor peso que tiene la capa de comunicación frente a la E/S. Los resultados muestran que para la arquitectura propuesta, con 5 nodos y con 32 procesos leyendo datos en paralelo cada uno, se obtiene 20 veces más volumen de comunicación de datos que E/S. Esto es debido que la red es un único recurso compartido por todos los nodos, mientras que la E/S, escala con el número de nodos, y se ve decrementada cuanto mayor es el paralelismo. La Figura 5.8 muestra como la E/S escala en a medida que el número de nodos aumenta.

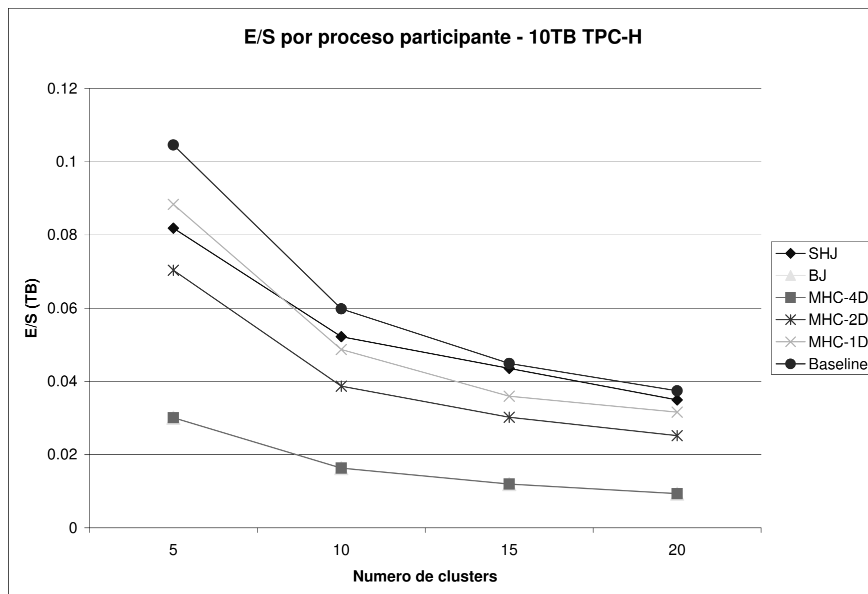


Figura 5.8 Escalabilidad de la E/S.

Recursos de memoria

La Figura 5.10-(a) muestra la cantidad de memoria requerida por SHJ para cada grupo de selectividades de la Tabla 5.1. Podemos observar que, para este caso en particular, SHJ necesita como máximo 1.8 GB de memoria principal con tal de almacenar todos los *bit filters* del sistema, que tienen una fracción de falsos positivos $P = 0.05$. La Figura 5.10-(b) muestra la relación entre la fracción de falsos positivos P, y la memoria requerida por un *bit filter* y su selectividad. De modo que, si SHJ estuviera limitado por la cantidad de memoria, entonces un incrementeo en el valor de

P permitiría que los *bit filters* cupieran en memoria. Usar menos memoria incrementa la fracción de falsos positivos, lo cual se puede realizar de forma controlada. Los grupos 3 y 4 son siempre los que más memoria requieren debido a que *orders* y *ps_partsupp*, que son las tablas de dimensión más grandes, tienen una selectividad muy alta, de modo que el tamaño de los *bit filters* creados durante sus respectivas fases de build es grande debido a la presencia de un gran número de valores distintos.

Si compararamos SHJ con BJ, la memoria requerida es ordenes de magnitud menos. Para una base de datos TPC-H de 10 TB, la mínima memoria requerida por BJ, un vector de bits con una entrada por tupla de la tabla de hechos, es de 7 GB de memoria principal. Y el índice de tipo *bitmap join* más pequeño, join entre $lineitem$ y $suppliers$, es de 70 TB, lo cual es 7 veces mayor que la misma base de datos.

5.5 Conclusiones

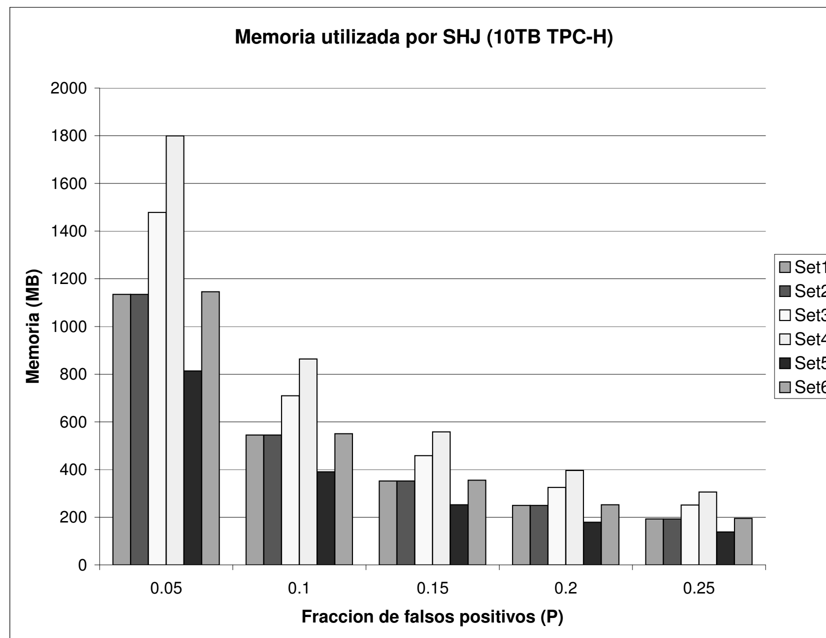
Consultas ad hoc. Join estrella					
	BJ	MHC	SHJ	PDBF	RBF$_B$
Aplicable a *Clusters*?	si	si	si	no	si
Ahorro comunicación	alto	bajo	alto	nulo	med.
Ahorro E/S tabla de hechos	alto	alto	bajo	bajo	nulo
Ahorro E/S operadores join	alto	bajo	alto	alto	nulo
Use of indices	alto	alto	nulo	nulo	nulo
Mem. usage	alto	bajo	med.	bajo	med.

Tabla 5.3 Características de cada técnica.

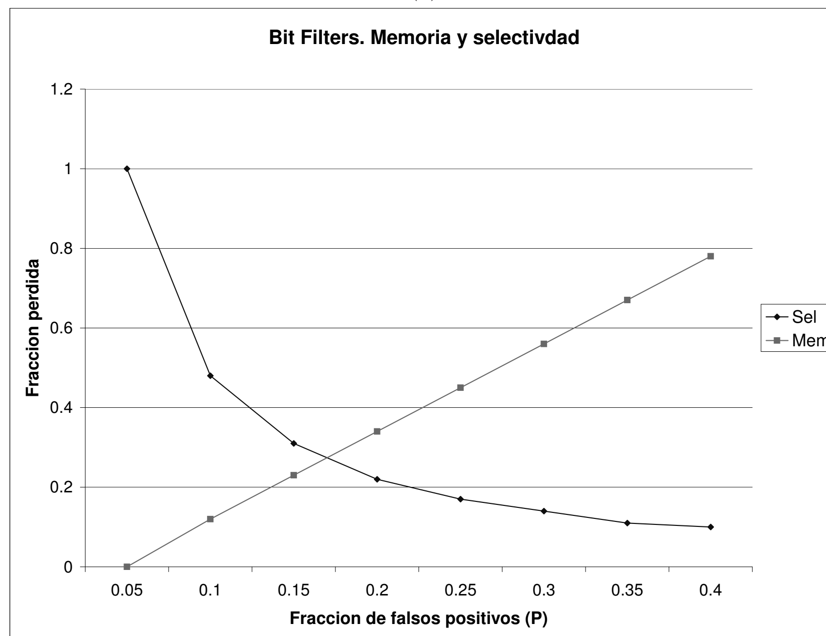
La Tabla 5.3 aporta un resumen de las principales características de cada una de las técnicas analizadas en este Capítulo para acelerar la ejecución de consultas ad hoc de tipo join estrella en arquitecturas *cluster*. El *bitmap join* (BJ) es la técnica que, desde un punto de vista teórico, realiza la ejecución ideal de un join estrella. Sin embargo, su implementación no es posible debido a los altos requerimientos tanto de memoria, como de mantenimiento de índices. Otras, técnicas como *Multi Hierarchical Clustering* (MHC), o la propuesta en este artículo, el *Star Hash Join* (SHJ), intentan implementar la misma idea que BJ de una forma más práctica, cada una con sus ventajas y desventajas.

SHJ es una generalización de los *Pushed Down Bit Filters* (PDBF) a través de la técnica *Remote Bit Filters* (RBF$_B$). Nuestro análisis demuestra que SHJ es la mejor alternativa para evitar comunicación de datos entre nodos, y los resultados se acercan al máximo teórico de BJ para un gran rango de posibles consultas. Además, no requiere de uso de índices, y no es agresivo en el uso de memoria. Por otra parte MHC, que no obtiene buenos resultados en cuanto a comunicación de memoria, sí que tiene un gran impacto en la E/S ahorrada en un amplio espectro de casos.

Nuestro análisis también demuestra que en arquitecturas *cluster*, uno de los principales cuellos de botella es la comunicación de datos. Las arquitecturas *cluster* alivian el problema de la E/S particionando los datos de forma horizontal, además, en el caso de configuraciones CLUMP, los datos, en cada nodo SMP, pueden estar estratégicamente particionados entre los discos de forma que los procesos accedan a ellos en paralelo. Sin embargo, la red es un sólo recurso a compartir por todos los nodo del sistema, convirtiéndose en un serio cuello de botella. De ahí que podamos



(a)



(b)

Figura 5.9 (a) Memoria utilizada por la técnica *Star Hash Join*; (b) Análisis de los bit filters. E/S realizada.

resalzar la importancia de SHJ, y proponerla como la técnica a utilizar para aliviar la carga de la capa de comunicación.

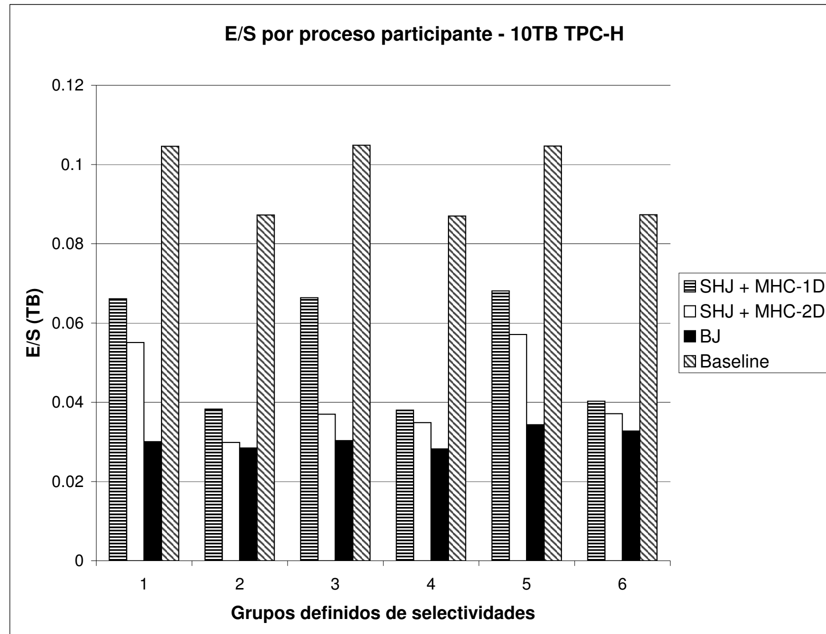
Solución híbrida

Por un lado tenemos SHJ, que es útil desde el punto de vista de la comuncicación de datos. Por otro lado, tenemos MHC, que es útil para el ahorro de la E/S. Concluimos el Capítulo proponiendo una solución híbrida que que consigue un buen balanceo entre el ahorro de comunicación y el de E/S. La Figura 5.10 muestra los resultados de la solución híbrida entre SHJ y MHC para los seis grupos definidos en la Tabla 5.1.

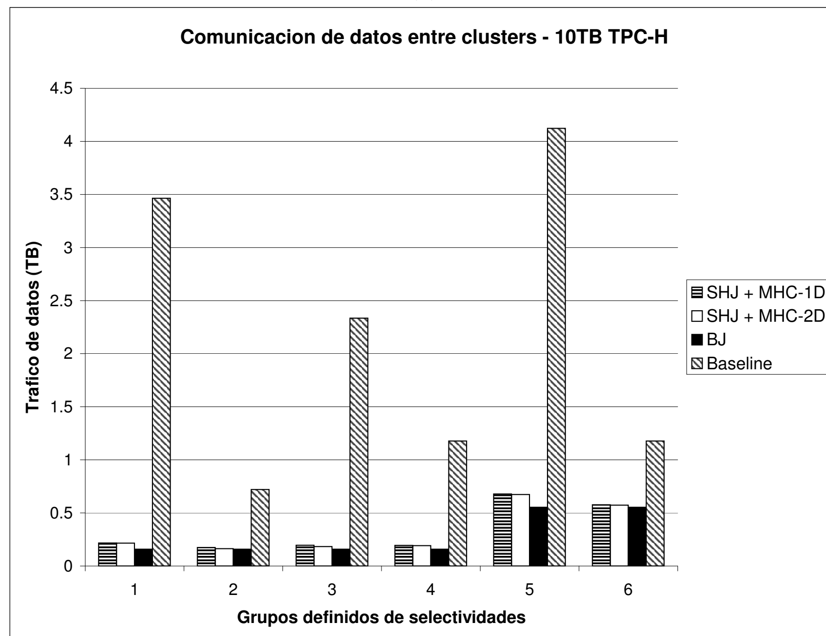
La Figura 5.10-(a) muestra la cantidad de comunicación de datos para cada técnica. Notar que no se muestra la combinación de SHJ con MHC-4D ya que, en este caso, MHC-4D obtiene, por sí sola, resultados óptimos. En la gráfica podemos observar que gracias a SHJ, se soluciona el problema de la comunicación de datos presente en MHC para consultas ad hoc, y que, en todos los casos se obtiene unos resultados muy próximos a los obtenidos por BJ.

La Figura 5.10-(b) muestra la cantidad de E/S realizada por cada técnica. En la gráfica podemos ver que se reduce la E/S de forma considerable en un 50% de los casos utilizados para nuestra evaluación.

Así pues, podemos considerar la solución híbrida como la mejor alternativa para obtener mejoras tanto en la capa de comunicación, como en la E/S.



(a)



(b)

Figura 5.10 Solución híbrida. (a) Comunicación de datos; (b) E/S realizada.

6

MEMORY CONSCIOUS HYBRID HASH JOIN

6.1 Introdución

Para obtener un buen rendimiento en arquitecturas *cluster* resulta indispensable sacar el máximo potencial de los bloques que la forman. Prueba de ello es la técnica *Star Hash Join* explicada en el anterior Capítulo, que mejora el rendimiento de las arquitecturas *cluster* gracias al uso de los *Pushed Down Bit Filters*, técnica, esta última, única y exclusivamente orientada a acelerar la ejecución de consultas multijoin en sistemas secuenciales o paralelos con recursos compartidos.

Este Capítulo se centra en la explotación de recursos en arquitecturas con multiprocesadores simétricos (SMP) durante la ejecución de la operación de join. El rendimiento de las arquitecturas SMP, es muy dependiente de la memoria compartida por los procesadores. De modo que, un uso eficiente de la memoria es realmente importante con el fin de minimizar la E/S, contención de memoria y, puntos de sincronización entre procesos. Todo esto hace que el balanceo de carga durante tiempo de ejecución, tanto dinámico como estático, sea de vital importancia en este tipo de arquitecturas. En este Capítulo nos adentramos en ello durante la ejecución de la operación de join en sistemas SMP.

6.1.1 Trabajo relacionado y motivación

Los algoritmos basados en hash explicados en [36] fueron extendidos a arquitecturas paralelas sin recursos compartidos en [32]. El *Hybrid Hash Join* fue paralelizado en la máquina de bases de datos relacional GAMMA en [33; 34]. En [77] se evalúan cuatro algoritmos paralelos para arquitecturas sin recursos compartidos. En este estudio se demuestra que el algoritmo *Hybrid Hash Join*, bajo cualquier contexto, obtiene mejor rendimiento que los algoritmos *Sort Merge Join* [44], el *Hash Join* [17], y el *GRACE Hash Join* [50]. Finalmente concluyen que cuando los valores de la clave de join de la relación de *build* tienen mucho sesgo, entonces, los algoritmos basados en hash no deberían de ser utilizados. El pobre rendimiento de los algoritmos basados en hash para datos no uniformemente distribuidos se demuestra en [80].

Todos estos trabajos previos se realizan bajo el contexto de arquitecturas sin recursos compartidos, las cuales muestran problemas de comunicación entre pocesos y de balanceo de carga cuando los datos presentan sesgo. Estos problemas son solventados con el uso de arquitecutas SMP, sin embargo, el precio a pagar se traduce en contención de memoria. El espacio de direcciones compartido facilita la tarea de trasladar algoritmos secuenciales a un contexto paralelo, siendo SMP

más fácil de usar que otras arquitecturas paralelas. El trabajo realizado relacionado con algoritmos basados en hash para configuraciones SMP no es tan amplio como el realizado para arquitecturas sin recursos compartidos.

En [54] (véase Capítulo 2, Sección 2.4.2) se realiza un modelo analítico para arquitecturas SMP de los algoritmos *GRACE Hash Join*, *Hybrid Hash Join* y *hashed loops join*. El modelo de algoritmo para el *Hybrid Hash Join* propuesto en este trabajo tiene como principal desventaja que, durante la fase de join del algoritmo, todos los procesos trabajan en paralelo sobre un mismo *bucket* y, en consecuencia, padece de contención de memoria y exceso de sincronización entre procesos. El mismo trabajo también elude el problema de *bucket overflows*, confiando que cada *bucket* resultado del particionado de la relación de *build* siempre cabrá en memoria.

Tal y como se explicó en el Capítulo 2 (Sección 2.4), el problema de contención de memoria es solventado en [63]. Esta propuesta, sin embargo, no es consciente de la memoria disponible por el algoritmo *Hybrid Hash Join*, y asume que cada *bucket* procesado siempre cabe en memoria. De esta forma, se elude nuevamente el problema de *bucket overflows* en caso de que el heap de memoria a utilizar por los procesos participantes en la operación de join esté acotado.

También existen trabajos que centran su investigación en diferentes recursos que se convierten en cuellos de botella durante la ejecución de joins en paralelo: esquemas de control de concurrencia para la contención de recursos de memoria [15; 21], planificación de recursos basados en la prioridad [20], estrategias para el manejo de la memoria de forma dinámica [16; 58; 69] y balanceo dinámico de carga para determinar el grado interno de paralelismo de una consulta [59; 72].

En este Capítulo nos centramos en algoritmos autoconfigurables para la ejecución del *Hybrid Hash Join* en arquitecturas SMP. Si observamos el trabajo previo realizado, podemos ver que hay varios métodos para paralelizar el algoritmo *Hybrid Hash Join* tanto para arquitecutras paralelas sin recursos compartidos, como para arquitecturas con recursos compartidos, SMP. Mientras las primeras están libres de contención de memoria, las segundas presentan una rápida comunicación entre procesadoros y un buen balanceo de carga. El objetivo de nuestro trabajo es encontrar un algoritmo autonómico que extraiga lo mejor de ambos mundos durante la ejecución del algoritmo HHJ paralelo en arquitecturas SMP.

6.1.2 Contribuciones

Proponemos el algoritmo parallelo *Memory-Conscious* para el *Hybrid Hash Join* (HHJ) consciente de los recursos de memoria disponibles en arquitecturas SMP. El algoritmo balancea dinámicamente la carga de trabajo cuando procesa los *buckets* almacenados en disco durante la fase de join del algoritmo HHJ. Los *buckets* son planificados dependiendo de la cantidad de memoria disponible, y del número de procesos asignados a cada *bucket* se decide dependiento del coste de procesamiento requerido, y de los recursos disponibles. De este modo, aportamos autonomía al SGBD [53; 74] permitiendo configuración de la carga del trabajo de la base de datos en tiempo de ejecución.

Nuestro algoritmo cumple los siguientes tres objetivos:

1. Reducir la contención de memoria y sincronización entre procesos balanceando los procesos participantes en la operación de join entre los *buckets* en función de la carga.

2. Reducir la E/S y el consumo de cpu a través de un buen uso de los recursos de memoria compartidos por los procesos participantes en la operación de join.

3. Introducir cualidades autonómicas al SGBD adaptando de forma dinámica el grado de paralelismo para cada *bucket*, y la cantidad de memoria compartida y necesitada por cada proceso participante.

Comparamos nuestra estrategia contra otras dos estrategias paralelas para el algoritmo HHJ. La primera se basa en asignar todos los procesos participantes a un mismo *bucket* [54], priorizando la rápidez de procesado por *bucket*. De forma opuesta está la alternativa que asigna un proceso por *bucket* cargado en memoria, priorizando de esta forma el *throughput*, y minimizando contención de memoria.

Analizamos en profundidad las tres estrategias:

- Hemos codificado un prototipo de los tres algoritmos usando DB2 UDB v8.1 y ejecutado nuestras pruebas en un nodo SMP de 8 vias.

- Hemos modelado la E/S, la contención de memoria y el consumo de cpu para las tres estrategias. También tenemos en cuenta la presencia de distribuciones no uniformes de datos.

- Tanto las ejecuciones reales como los modelos, muestran que, en la mayoría de los casos, nuestra estrategia mejora respecto con las otras dos estrategias.

6.2 Algoritmos paralelos del HHJ para SMP

Como se ha explicado en el Capítulo 2 (Sección 2.4.2), y se detalla en [54], las fases de *build* y *probe* del algoritmo HHJ son fáciles de paralelizar para arquitecturas SMP. En este Capítulo nos centramos en la paralelización de la tercera fase, la fase de join, que es más difícil de paralelizar y que, cuando el algoritmo HHJ no se puede ejecutar en memoria, puede llegar a ser la fase de más coste computacional y con más E/S.

Para el resto del Capítulo se emplea la misma nomenclatura a la utilizada en el Capítulo 2 (Sección 2.4), donde detallamos el algoritmo HHJ. Se recomienda su lectura para una mejor comprensión.

Como primera alternativa, propuesta en [54] y detallada gráficamente en la Figura 6.1, cada par de *buckets* $\langle R_i, S_i \rangle$ se procesa en paralelo por todos los procesos participantes en la operación de join. Esta estrategia requiere de sincronización entre procesos, y puede repercutir en un incremento de la contención de memoria. Utilizamos esta estrategia como base (*Basic*) a lo largo del Capítulo.

Como segunda implementación, y que podría ser considerada como una extensión a arquitecturas SMP de lo que es el paralelismo en arquitecturas sin recursos compartidos, cada par de *buckets* $\langle R_i, S_i \rangle$ son procesados en paralelo y de forma individual por cada proceso participante en la operación de join, tal y como se muestra en la Figura 6.1. Esta estrategia evita contención de memoria y sincronización entre procesos. Denominamos a esta iplementación del algoritmo HHJ como *Non Memory-Conscious* (NMC).

Esta segunda implementación requiere de una pequeña modificación durante la fase de *build* del algoritmo HHJ. En este caso, la relación de *build* R debe de ser particionada teniendo en cuenta que el número de páginas disponible $|M|$ permite disponer un *bucket* R_i por proceso durante la fase de join del algoritmo HHJ, ver Figura 6.1. Así pues, modificamos el método por el que B es calculado (ecuación 2.4, Capítulo 2) e introducimos conocimiento acerca del número de procesos p involucrados en la operación de join:

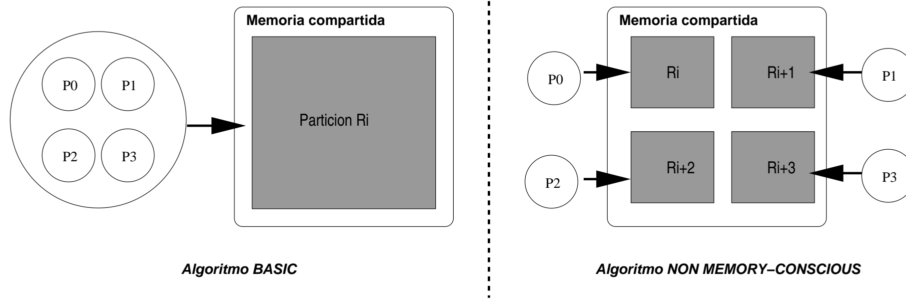


Figura 6.1 Implementaciones *Basic* y NMC del algoritmo HHJ. Asumimos 4 procesos y i=1..B.

$$B = max\left(0, \frac{|R| \times F - |M|}{|M| - p}\right) \times p \tag{6.1}$$

De esta forma, asumiendo una distribución uniforme de los datos, el tamaño de cada *bucket* se calcula como $|R_i| = \frac{|R|}{B}$, y requiere de una cantidad de memoria $|M_i| = \frac{|M|}{p}$ para ser procesado. El principal problema de esta implementación es que el número de *buckets* incrementa de forma proporcional a p. Entonces, al estar B limitado por M, puede ocurrir que, debido a un consumo agresivo de memoria, se produzcan *bucket overflows* durante el procesado de p *buckets* en paralelo.

6.3 El algoritmo Memory-Conscious

Proponemos el algoritmo *Memory-Conscious* (MC) para la fase de join del algoritmo HHJ. Este algoritmo solventa los problemas de los algoritmos paralelos explicados en la Sección previa. Por una parte, soluciona el problema de contención de memoria del algoritmo *Basic*. Por otra parte, reduce el peligro de la aparición de *bucket overflows* del algoritmo Non Memory Conscious.

6.3.1 El algoritmo

Durante el resto del Capítulo, definimos $|M_i|$ y p_i como la cantidad de memoria compartida (en páginas), y la cantidad de procesos asignados a un determinado par de *buckets* $\langle R_i, S_i\rangle$.

De una forma similar a como lo hace el algoritmo NMC, ejecutamos la fase de particionado basándonos en la ecuación 6.1. De esta forma siempre se intenta cargar en memoria un *bucket* por proceso participante, evitando contención de memoria.

Cuando alcanzamos la fase de join del algoritmo HHJ, entonces MC construye una lista ordenada de todos los *bucket* R_i, ordenados de mayor a menor. De este modo, el algoritmo MC se adapta de forma dinámica a la memoria compartida por los procesos participantes en la operación de join, cargando tantos *buckets* en memoria como sea posible en orden descendiente. Así pues, la memoria usada por cada *bucket* R_i es:

$$|M_i| = min(|M|, |R_i| \times F)$$

de esta forma se evitan *bucket overflows* a no ser que sea forzoso porque $|R_i| \times F > |M|$.

El algoritmo empieza cargando los primeros n *buckets* más grandes en memoria ($n \le B \wedge n \le p$), y balancea los procesos entre ellos. Siendo LR la lista de los R_i *buckets* cargados en memoria, podemos calcular el número inicial de procesos por *bucket* (p_i) de la siguiente forma:

for R_i en LR **do** $p_i = 1$; $p_disponibles = p - n$;
while $(p_disponibles)$ **do**
{
/* *Encontramos el bucket con mas carga a partir del ratio*$(\frac{|R_i|}{p_i})$*/
$max_cargado = i : Ri \in LR : \left(\forall k : \frac{|R_i|}{p_i} \geq \frac{|R_k|}{p_k}\right)$
$p_{max_cargado++}$; $p_disponibles - -$;
}

Así pues, inicialmente tenemos n grupos de procesos, uno por cada *bucket* cargado en memoria. Los *buckets* más grandes en tamaño son procesados al principio y, los procesos son planificados a medida que los *buckets* van disminuyendo de tamaño. Una vez que un grupo de procesos ha finalizado el join entre un *bucket* R_i con su correspondiente *bucket* S_i de la relación de *probe* almacenado en disco, entonces se procede a tratar con los siguientes $B - n$ *buckets*, hasta que sean agotados.

En caso de que un grupo de procesos pueda cargar más de un *bucket* en la memoria que estaban utilizando, entonces, los procesos de ese grupo son planificados de nuevo como en la inicialización. Y por otro lado, si se da el caso de que no se necesita toda la memoria que los procesos estaban utilizando, entonces, se libera de forma que pueda ser utilizada por los siguientes *buckets* a ser procesados.

Notar que el hecho de ordenar los *buckets*, sirve para saber que los futuros *buckets* van a ser iguales o más pequeños, de modo que podemos planificar los procesos de forma fácil ya que sólo hay que separarlos del grupo inicial en caso de que fuera necesario.

A modo de ejemplo, podemos mirar la Figura 6.2 donde tenemos nueve *buckets* y cuatro procesos disponibles. Inicialmente, solo R_1 y R_2 caben en memoria, así que empezamos teniendo dos grupos, con dos procesos asignados a R_1 y R_2 respectivamente. Digamos que el procesado del join del par $\langle R_1, S_1\rangle$ finaliza antes que el join en el par $\langle R_2, S_2\rangle$. Entonces, sólo queda espacio para cargar R_3, de modo que los procesos P_0 y P_1 permanecen juntos trabajando sobre R_3. Por otro lado, cuando el join del par $\langle R_2, S_2\rangle$ finaliza, hay espacio en memoria para cargar R_4 y R_5, de modo que, los procesos P_2 y P_3 son reasignados uno a cada *bucket* respectivamente. Si estos dos procesos finalizan antes que el procesado del join entre el par $\langle R_1, S_1\rangle$, entonces tendrán que proceder con los *buckets* R_6 y R_7, y cuando P_0 y P_1 finalicen con el join del par $\langle R_3, S_3\rangle$, entonces, serán reasignados a R_8 y R_9.

En el caso particular de que los datos estén uniformemente distribuidos, todos los *buckets* tendrán el mismo tamaño. De este modo, siempre cargaremos el mismo número de *buckets* en memoria y cada *bucket* tendrá que tener el mismo número de procesos asignados a él. En este caso, si es posible cargar en memoria un *bucket* por proceso participante, entonces MC se comporta igual que NMC. Sin embargo, si esto no es posible, entonces, el algoritmo MC se acerca al algortimo *Basic* e intenta minimizar el riesgo de *bucket overflows* cargando exactamente el número de *buckets* que caben en memoria.

6.3.2 Propiedades autonómicas

El hecho de que la carga de las bases de datos está creciendo en escala y complejidad ha motivado la necesidad, por parte de los SGBDs, de desarrollar mecanismos de autosuficiencia o autonomía [53; 74; 86].

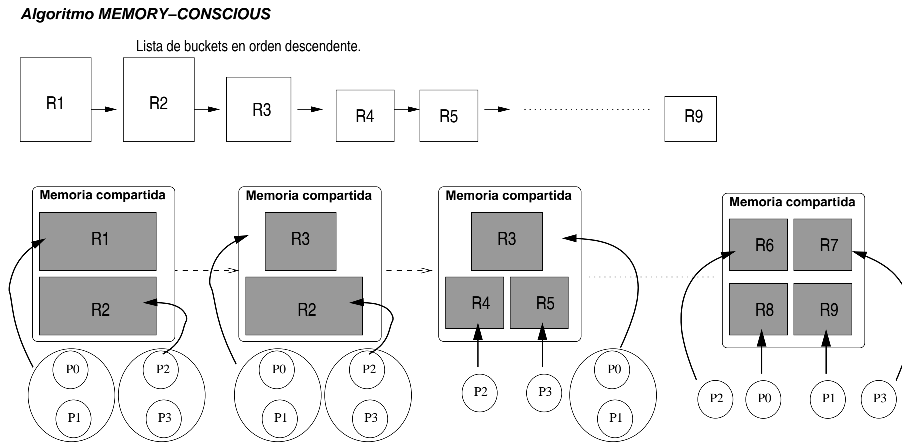


Figura 6.2 El algoritmo *Memory-Conscious*.

La computación autonómica se debe de anticipar a las necesidades del sistema y ajustar a las variantes circunstancias. Nuestro algoritmo contribuye en este aspecto y facilita propiedades autonómicas al algoritmo HHJ:

- El grado de paralelismo para cada *bucket* es autoconfigurado en función de la carga.
- La memoria compartida por los procesos participantes es autoadministrada, reduciendo el riesgo de *bucket overflows* y desuso de memoria.
- La memoria no utilizada puede ser liberada para otras operaciones en el SGBD u otras aplicaciones ejecutadas en el servidor.
- El algoritmo MC puede pedir más memoria si es necesario para evitar *bucket overflows*.

En trabajos previos al nuestro, el particionado para algoritmos basados en hash es dimensionado de forma estática [36; 50]. Poca investigación se ha realizado relacionado con la autonomía para la planificación de particiones. Por ejemplo, en [85] esquemas de memoria dinámica son propuestos para las operaciones hash join. Sin embargo, el particionado es decidido por el optimizador y puede causar *bucket overflows*. En [30], se proponen estrategias que varían dinámicamente el tamaño de las particiones, así como el balanceo dinámico de carga para determinar el grado de paralelismo en una consulta [59; 72]. Sin embargo, el primer trabajo no contempla el problema de la contención de memoria, ni el de planificación de procesos. Por otro lado, el segundo trabajo no afronta operaciones de carácter autonómico como el propuesto en este Capítulo.

6.4 Configuración de la evaluación

Hemos introducido un prototipo de los algoritmos paralelos MC, NMC y *Basic* dentro del algoritmo *Hybrid Hash Join* codificado en el Sistema Gestor de Bases de Datos DB2 UDB v8.1. Para nuestras pruebas utilizamos un nodo SMP IBM p660s, con 8 Power RS64-IV compartiendo un total de 16GB de memoria RAM. El sistema operativo es AIX versión 5.1. En su defecto, ejecutamos nuestros tests con un grado de paralelismo igual a 8, de modo que utilizamos todos los procesadores.

Para la ejecución de nuestras pruebas la máquina se ha reservado de forma exclusiva, es decir, en tiempo de ejecución el sistema no está siendo compartido con ninguna otra aplicación.

Entorno de ejecución

Hemos utilizado una base de datos TPC-H con un tamaño de 64GB. La consulta utilizada para la evaluación es similar a las consultas de TPC-H, y se ha creado para ilustrar el problema y entender las características de las estrategias evaluadas. Básicamente, la consulta consiste en un hash join entre las relaciones *orders* y *partsupp*. La relación *orders* y *partsupp* juegan el papel de S (relación de *probe*) y R (relación de *build*) respectivamente. También hay una restricción sobre *partsupp* con el objetivo de variar el tamaño de la relación R.

El modelo analítico

En el Apéndice C se presenta un modelo analítico para los tres algoritmos paralelos evaluados. En la siguiente Sección también se presentan resultados para el modelo analítico que se comparan con los resultados reales con el fin de validar el modelo.

El modelo nos ayuda a explicar los resultados reales a partir del modelo de contención de memoria para cada algoritmo, y su validación para distribuciones uniformes nos permite dar una idea de cómo cada algoritmo se podría comportar ante la presencia de distribuciones de datos con sesgo.

6.5 Resultados

Asumiendo que por defecto los datos son uniformes, realizamos nuestra evaluación en tres contextos diferentes dentro del procesado de un join:

1. E/S intensiva

2. Uso intensivo de la memoria donde la presencia de *bucket overflows* es más bien escasa.

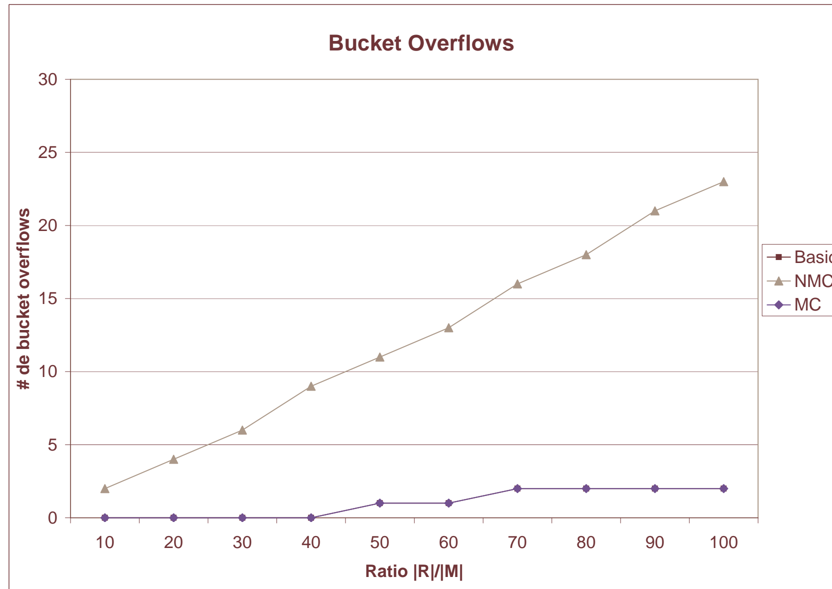
3. Distribución no uniforme de los datos.

Para cada uno de los tres contextos mostraremos (i) los resultados reales obetnidos a partir de nuestras ejecuciones, y (ii) los resultados obtenidos por nuestro modelo analítico.

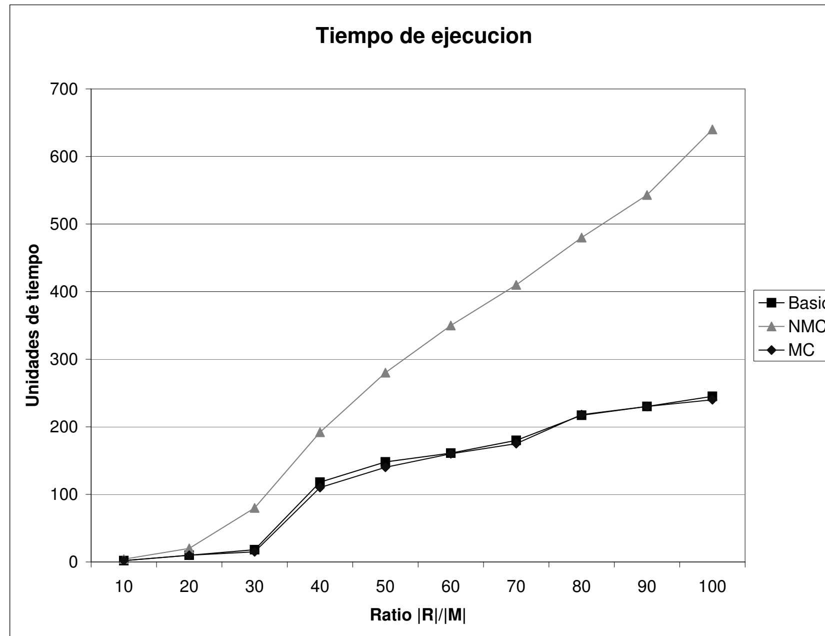
Consultas con join de gran procesado de datos. E/S intensiva

El objetivo de este análisis es mostrar el comportamiento de cada algoritmo cuando el principal cuello de botella es la E/S. En la Figura 6.3-(a) se muestra el número real de *bucket overflows* para diferentes tamaños de R, desde 10 a 100 veces el tamaño de la memoria asignada al algoritmo HHJ. NMC, debido al uso agresivo de la memoria muestra un número muy elevado de *bucket overflows*. Sin embargo, MC, al ser consciente de la cantidad de memoria disponible y del tamaño de los *buckets*, se aproxima al algoritmo *Basic* y sólo empieza a generar *bucket overflows* cuando $|R| \gg |M|$. Los tiempos de ejecución obtenidos durante la fase de join del HHJ se muestran en la Figura 6.3-(b), y están en consonancia con los resultados mostrados en la Figura 6.3-(a). NMC es el algoritmo más costoso, mientras que MC y el *Basic* se comportan de forma similar.

En la Figura 6.4 mostramos los resultados obtenidos por el modelo analítico. Podemos observar que tanto la E/S, como el tiempo de cpu tienen la misma tendencia que los resultado reales obtenidos para el tiempo de ejecución.



(a)



(b)

Figura 6.3 (a) Número de *bucket overflows*; (b) Tiempo de ejecución de la fase de join del algoritmo HHJ.

Uso intensivo de la memoria

En este segundo escenario analizamos los tres algoritmos cuando la cpu se convierte en el cuello de botella, y la contención de memoria es crucial.

La Figura 6.5, basándonos en la ecuacion C.1 del modelo analítico, muestra como la contención de memoria afecta a cada algoritmo para diferentes tamaños de la relación de *build* R. La gráfica

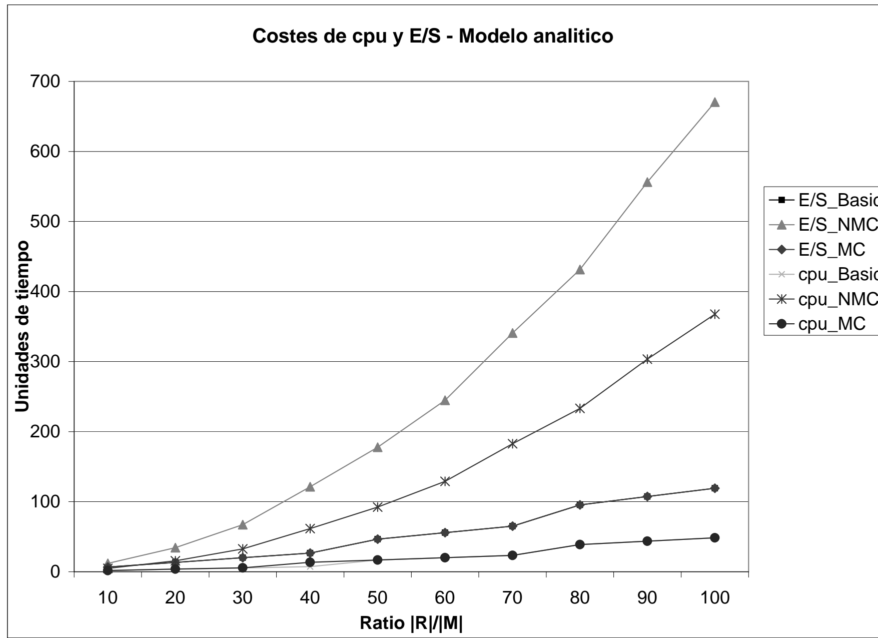


Figura 6.4 Resultados para la fase de join cuando ocurren *bucket overflows*.

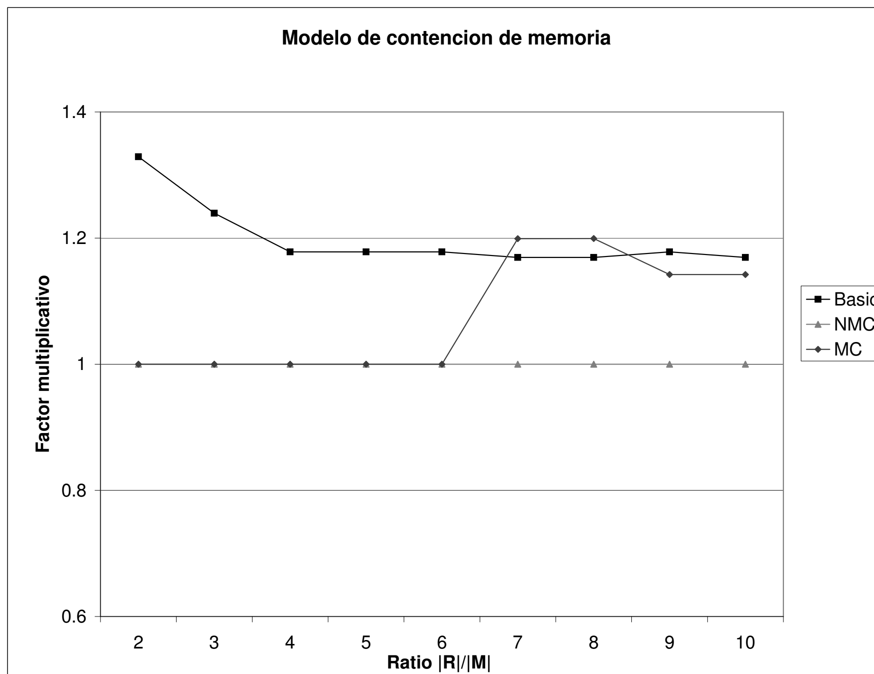


Figura 6.5 Contención de memoria.

muestra que el algoritmo MC está libre de contención de memoria hasta que la relación R empieza a ser lo suficientemente grande como para que no sea posible cargar un *bucket* por proceso en memoria. Llegado este caso, el número de procesos por *bucket* (p_i) es mayor que 1, de modo que el algoritmo MC empieza a tener contención de memoria, y se comporta de forma similar a la implementación *Basic*. También podemos observar que existen algunos pocos casos en los que MC puede tener mayor contención de memoria que el algoritmo *Basic*. Esto se debe al hecho de que en estos casos, el algoritmo MC puede tener más de un *bucket* por proceso en paralelo, de modo que, hay una pequeña franja en la que puede darse alguna combinación de valores para $|M_i|$, $|R_i|$ y p_i tal que son diferentes a los valores del algoritmo *Basic*, y que en estos casos se incremente la probabilidad de tener dos claves mapeadas en la misma página. Por el contrario, notar que para estos casos, el algoritmo MC tendrá menos sincronización entre procesos al estar éstos repartidos entre varios *buckets*. La sincronización de procesos no está modelada, pero existe desde el momento en que i) un proceso no puede leer el *bucket* S_i hasta que todas las tuplas del *bucket* R_i hayan sido introducidas en la tabla de hash, y ii) un proceso no puede empezar a procesar un nuevo *bucket* hasta que todos los procesos hayan finalizado con el *bucket* en tratamiento.

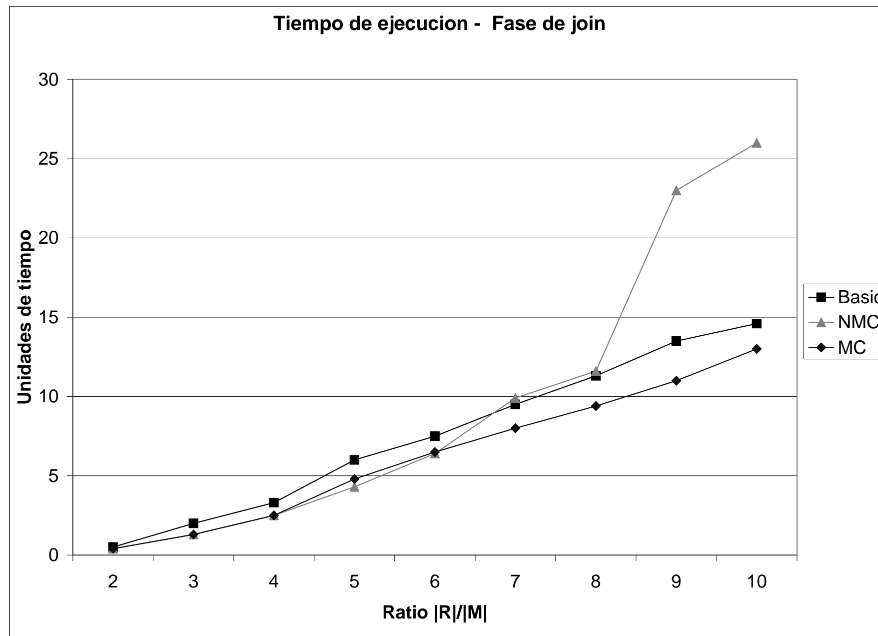


Figura 6.6 Tiempo de ejecución de la tercera fase del HHJ, la fase de join.

La Figura 6.6 muestra los tiempos de ejecución obtenidos durante la fase de join del algoritmo HHJ. En la gráfica podemos observar que MC y NNC mejoran el rendimiento del algoritmo *Basic*: para un ratio $\frac{|R|}{|M|} = \{1..5\}$ MC y NMC son capaces de cargar en memoria un *bucket* por proceso, quedando así libres de contención de memoria. Cuando el ratio $\frac{|R|}{|M|}$ alcanza un valor mayor o igual a 7, entonces, NMC empieza a generar *bucket overflows* y empieza a decaer su rendimiento. Por otra parte, el algoritmo MC adapta dinámicamente su comportamiento, y carga menos *buckets* balanceando los procesos entre los *buckets* en memoria. En la Figura 6.7 mostramos como cada algoritmo escala cuando no hay *bucket overflows* para ninguno de ellos. Como era de esperar, MC

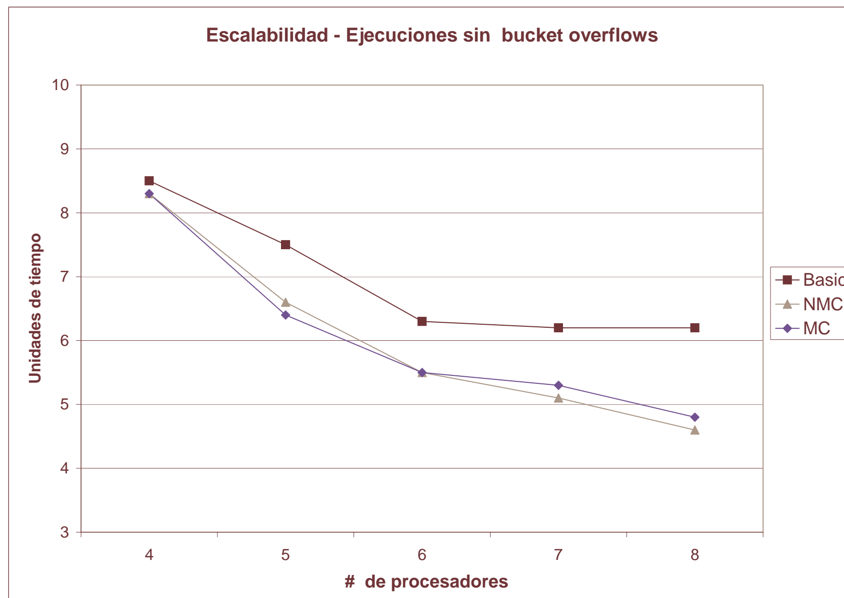


Figura 6.7 Escalabilidad. Tiempos de ejecución.

y NMC escalan mejor que el algoritmo *Basic* debido al hecho que cada procesador puede correr en paralelo sin contención de memeoria y/o sincronización entre procesos.

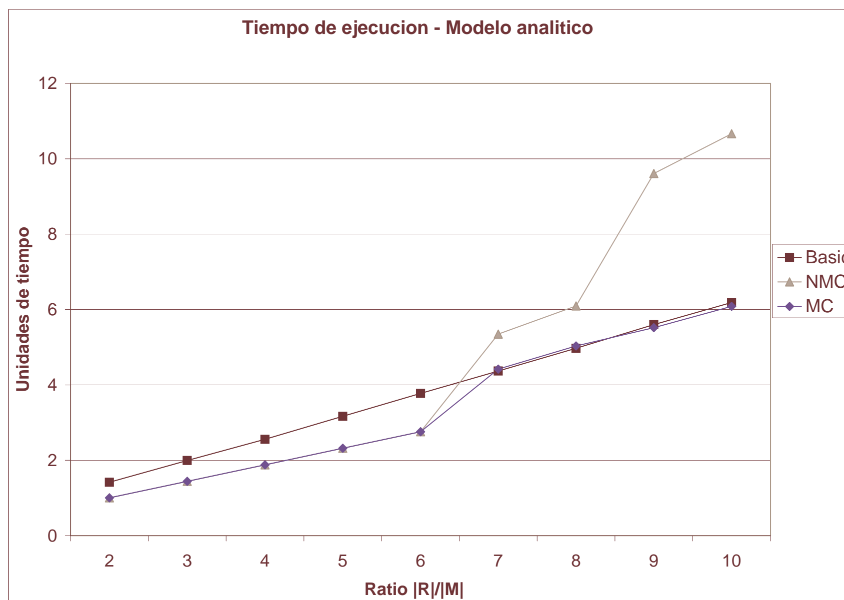


Figura 6.8 Tiempo de ejecución de la tercera fase del HHJ, la fase de join.

La Figura 6.8 muestra el modelo de costes para la cpu. No se muestra la E/S ya que es la misma para los tres algoritmos. De nuevo, el modelo matemático muestra las mismas tendencias

que los resultados reales. Las pequeñas diferencias existentes se pueden achacar al hecho que la sincronización entre procesos no esté modelada: podemos observar que el algoritmo MC no mejora el rendimiento del algoritmo *Basic* cuando más de un proceso trabaja en un sólo *bucket*.

Distribuciones no uniformes (sesgo en los datos)

A través del modelo analítico validado a lo largo del Capítulo, analizamos una distribución de datos no uniformes por un valor, aunque los resultados pueden ser generalizados para sesgo de datos en múltiples valores, teniendo de esta forma varios *buckets* de diferentes tamaños. Notar que el modelo matemático no ha sido validado para distribuciones no uniformes, sin embargo, y pese a sus limitaciones, analizamos los resultados esperados para distribuciones datos con sesgo.

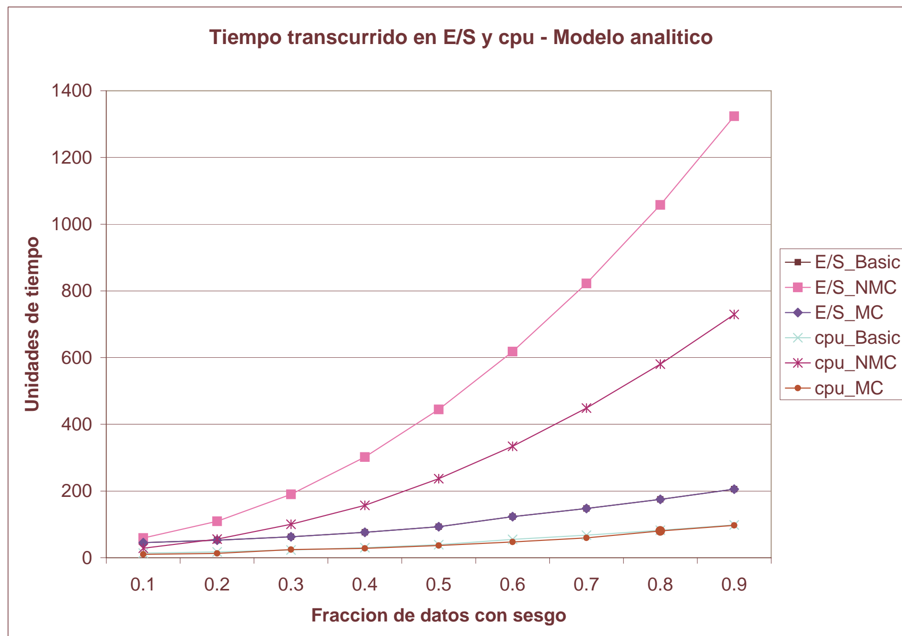
Los costes de cpu y E/S son mostrados en la Figura 6.9-(a). Podemos ver que, cuanto mayor es la fracción de datos con sesgo, mayor es el número de *bucket overflows* para el algoritmo NMC y, en consecuencia, obtiene peor rendimiento que las implementaciones MC y *Basic*. La contención de memoria se muestra en la Figura 6.9-(b). Podemos observar que MC obtiene los mejores resultados: con el objetivo de evitar *bucket overflows*, MC asigna un mayor número de procesos para ejecutar el *bucket* de mayor tamaño generando contención de memoria en este caso. Por otra parte, el procesamiento de los *buckets* más pequeños carece contención de memoria al ser ejecutados de forma individual por un solo proceso. En la Figura 6.9-(a) no se puede apreciar la mejora que MC obtiene sobre el algoritmo *Basic* debido a que el coste de procesar lo *buckets* de menor tamaño es despreciable cuando se compara con el *bucket* con datos sesgados.

6.6 Conclusiones

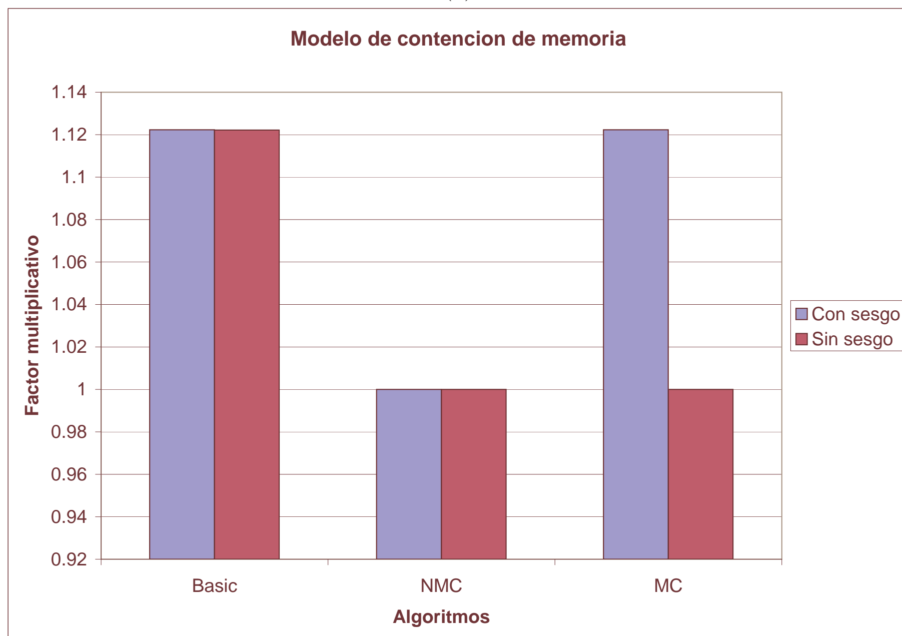
Existen varias estrategias para paralelizar el algoritmo *Hybrid Hash Join* en arquieturas con multiprocesadore simétricos. Hemos contemplado dos implementaciones base, la *Non Memory-Conscious* (NMC), la cual es equivalente a la propuesta para arquitecturas paralelas sin recursos compartidos, y la *Basic*, previamente propuesta en la literatura para arquitecturas paralelas con recursos compartidos. La primera tiene la ventaja de que esta libre de contención de memoria, mientras que la última presenta un buen balanceo de carga entre procesos. Nuestra propuesta, el algoritmo *Memory-Conscious* (MC), extrae lo mejor de los dos propuestas anteriores siempre mejorando el rendimiento respecto uno u otro dependiendo del contexto.

Cuando la memoria disponile es lo suficientemente grande para particionar la relación de *build* de forma que cada proceso trabaje con un sólo *bucket*, entonces el algoritmo MC está libre de contención de memoria y mejora respecto el algoritmo *Basic*. Por otro lado, cuando ocurren *bucket overflows*, entonces el algoritmo MC mejora respecto la implementación NMC, y balancea la carga entre los *buckets* que caben en memoria y los procesos disponibles. Para nuestras pruebas, cuando la E/S es intensa, el algoritmo NMC puede llegar a generar un número de *bucket overflows* un orden de magnitud mayor que el algoritmo MC.

También adaptamos el algoritmo MC con el fin de afrontar distribuciones no uniformes de los datos. En este caso, MC evita i) *bucket overflows* balanceando procesos entre los *buckets* con sesgo en los datos y, ii) contención de memoria cuando se procesan los *buckets* de menor tamaño replanificando los procesos de forma que trabajan individualmente en *buckets* separados.



(a)



(b)

Figura 6.9 (a) Costes de E/S y cpu de cada algoritmo bajo los efectos de sesgo en los datos; (b) Contención de memoria.

7

SECUENCIAS TEMPORALES DE DATOS. LOS DYNAMIC COUNT FILTERS

7.1 Introducción

En los anteriores Capítulos hemos centrado nuestro trabajo en la mejora del rendimiento de los Sistemas Gestores de Bases de Datos sobre arquitecturas paralelas. En este Capítulo, nuestro foco de atención es otra area de gran importancia dentro del procesamiento de grandes volumenes de datos, las secuencias temporales de datos, también denominadas *streams* de datos.

La calidad de servicio es un requerimiento muy importante en muchas aplicaciones relacionadas con *streams* de datos. Con el fin de cumplir con esta necesidad, se requiere de estructuras de datos que realicen un uso compacto de la memoria, y presenten rápidos métodos de acceso tanto de lectura como de escritura. En este Capítulo presentamos *Dynamic Count Filters* (DCF) y su versión particionada (*Partitioned* DCF). Ambas propuestas consisten en sus estructuras de datos y sus métodos de acceso, que como se demostrará a lo largo del Capítulo aportan una alta calidad de servicio.

7.1.1 Trabajo relacionado y motivación

Hoy en dia son habituales aplicaciones que manipulan grandes volúmenes de *streams* de datos provenientes de diferentes fuentes. Entornos de redes, donde la cantidad de comunicaciones crece con la popularidad de Internet, o servicios financieros orientados a la predicción de sucesos, son claros ejemplos.

Con el fin de satisfacer las necesidades de este tipo de aplicaciones, se necesita de estructuras que den soporte a la inserción y borrado de elementos pertenecientes a un conjunto de datos, y que rápidamente puedan dar una respuesta relativa a la cantidad de ocurrencias de un elemento cualquiera dentro del conjunto de datos manipulado por la aplicación.

En [40] se proponen los *Count Bloom Filters* (CBF), que extienden los *bit filters* de forma que sean capaces de contar el número de elementos que coinciden en una misma posición. En este estudio, cada *proxy* almacena un resumen del directorio de *cache* de cada *proxy* participante. Cada resúmen se representa a través de un CBF, y cada proxy se encarga de mantener el resumen de su memoria *cache*: cada vez que se inserta una dirección en la memoria *cache* de un *proxy*, se incrementan los respectivos contadores del CBF local, y se decrementan cuando dicha dirección se reemplaza de la memoria *cache*. Cada *proxy* envía su resumen local al resto de *proxies* periódicamente. De esta forma, cada *proxy*, antes de enviar ninguna consulta preguntando por una dirección en concreto,

chequea estos resúmenes con el fin de detectar qué *proxies* tienen una mayor probabilidad de responder. Con ello, CBF ayuda a ahorrar comunicación innecesaria de datos requiriendo un coste de memoria relativamente bajo en cada servidor *proxy*.

El principal problema de CBF es que los contadores tienen un tamaño estático, y la estructura no se adapta a determinados cambios en los datos, como distribuciones con sesgo, que pueden llegar a saturar los contadores. Una vez los contadores se saturan, quedan inutilizados, de forma que no pueden aportar información alguna, perdiendo precisión y desaprovechando espacio de memoria.

Los CBF también han sido investigados para su uso en entorno de redes para resumir el contenido de nodos en sistemas *peer-to-peer* [19]. En [51] se aplican para reducir el número de búsquedas de nombres en sistemas distribuidos. Luego en [37] se presenta un nuevo algoritmo en el que los CBF se utilizan para acelerar el encaminamiento de direcciones IP en *routers*.

Con el fin de resolver la falta de adaptabilidad y precisión de los CBF, los filtros necesitan de soluciones dinámicas que permitan expander sus contadores a través de una operación que denominamos operación de *rebuild*. Cohen y Matías [27], proponen una representación dinámica de los CBF, los *Spectral Bloom Filters* (SBF). Los SBF solucionan la falta de adaptibilidad de los CBF, donde cada contador adquiere un tamaño de bits variable según el número de ocurrencias mapeadas en cada contador. Sin embargo, los SBF, con el fin de adquirir este grado de adapatibilidad, necesitan de estructuras de índices para acceder a los contadores. Esto hace que los accesos tanto de escritura como de lectura sean más lentos en comparación con los CBF, cuyos accesos son directos debido a que todos los contadores tienen el mismo número de bits.

El trabajo propuesto por Cohen y Matías, a parte de proveer una nueva estructura de datos, tambien presenta nuevos algoritmos para reducir la probabilidad y la magnitud de errores. Los histogramas Bloom [81] son una visión comprimida de los SBF, y se utilizan para mantener estadísticas aproximadas para los *path* en datos XML.

En este Capítulo presentamos dos nuevas generaciones de los CBF: los *Dynamic Count Filters* (DCF), y su versión particionada *Partitioned Dynamic Count Filters* (PDCF). En la Tabla 7.1 se muestra una comparativa de las características principales de las cuatro técnicas mencionadas. Podemos observar que DCF adquiere las buenas propiedades de sus predecesoras: el rápido acceso de los CBF y la adaptabilidad de los SBF. Además los DCF nunca saturan sus contadores, siendo capaces de representar *streams* ilimitados de información, o adaptarse a circunstancias extremas en las que los datos tienen mucho sesgo. Sin embargo, aunque los DCF son más eficientes que los SBF y los CBF en cuanto a consumo de memoria y tiempo de *rebuild* de la estructura, estos dos aspectos son mejorados por su versión particionada, los PDCF. Esta mejora en el tiempo de *rebuild* y el uso de la memoria se transforma en una mejor calidad de servicio por parte de los PDCF.

	Tamaño de contadores	Tiempo de acceso	Coste de un *rebuild*	Saturación de contadores	Desaprovechamiento de memoria
CBF	estático	rápido	n/a	si	alto
SBF	dinámico	lento	alto	posible	muy alto
DCF	dinámico	rápido	muy alto	no	alto
PDCF	dinámico	rápido	bajo	no	moderado

Tabla 7.1 Comparación cualitativa de CBF, SBF, DCF y PDCF.

7.1.2 Contribuciones

Podemos enumerar las contribuciones de este Capítulo como sigue:

- La propuesta de los *Dynamic Count Filters* (DCF), y de su versión particionada, los *Partitioned Dynamic Count Filter* (PDCF), con una detallada explicación de sus estructuras y métodos de acceso.
- Un modelo analítico que nos permite encontrar el número óptimo de particiones para los PDCF.
- Un modelo de costes asintóticos de los SBF, los DCF y los PDCF para las diferentes operaciones de inserción, borrado, consulta, y *rebuild*.
- Una comparación detallada entre las tres representaciones dinámicas de los CBF: SBF, DCF y PDCF.
- Una exhaustiva evaluación de SBF, DCF y PDCF.

Los resultados de la ejecución de las diferentes operaciones para distintos escenarios, usando DCF, PDCF y SBF como representaciones de los datos, muestra que (i) los DCF, siendo el doble de rápidos que los SBF, consumen la mitad de memoria, y que (ii) los PDCF consumen la mitad de memoria comparado con los DCF, y presentan un thoughput un orden de magnitud mayor que los DCF gracias a la mejora en el tiempo de *rebuild* de la estructura.

7.2 Count Bloom Filters (CBF)

Un *Count Bloom Filter* (CBF) consiste, básicamente, en un *bit filter* en el que cada bit correspondiente a cada una de sus m entradas se substituye por un contador de bits, $C_1, C_2, ..., C_m$ contadores. Así pues, un CBF representa un conjunto de n elementos $S = s_1, ...s_n$, dónde cada elemento puede presentar valores repetidos, siendo el total de valores representado $M = \mu n$. De forma similar a un bit filter, cada vez que se inserta un elemento s en el conjunto de datos representado, d funciones de hash se utilizan para actualizar d entradas del filtro $(h_1(s), h_2(s), ..., h_d(s))$. Mientras que en el bit filter, las entradas serían simplemente actualizadas a 1, en el CBF el contador en cada una de las d entradas se incrementa en uno. De forma análoga, cuando un elemento s se borra del conjunto de datos representado, los contadores en las mismas d entradas se decrementan en una unidad. En todo momento, la suma de todos los contadores del CBF es igual a $\sum_{j=1..m} C_j = d \times M$.

La representación común de un CBF es una estructura de datos en la que los contadores tienen un tamaño fijo. Dicha representación tiene dos inconvenientes de gran importancia:

1. Dado el momento en el que la inserción de un elemento cause la saturación de un contador por *overflow*, entonces, no podrán llevarse a cabo las operaciones de borrado mapeadas en el mismo contador.
2. Esta representación no es óptima ya que todos los contadores tienen el mismo tamaño, desperdiciando así espacio de memoria.

En [37] se referencía el problema de la saturación de contadores. En este trabajo, se reconstruye de nuevo toda la estructura con un tamaño mayor una vez el número de contadores saturados supera un cierto límite. Esta reconstrucción resulta muy costosa, ya que todos los mensajes deben

de volver a ser reinsertados de nuevo. Además, la estructrua queda inutilizada durante un largo período de tiempo ya que es inaccesible durante la operación de reconstrucción.

7.3 Spectral Bloom Filters

Cohen y Matías [27] propusieron los *Spectral Bloom Filters* (SBF), que son una representación compacta de los CBF. El principal objetivo de un SBF es que los contadores utilicen el mínimo de bits necesario para representar el contaje del número de elementos mapeados en cada posición del filtro. Un SBF consiste en un vector base compacto de $C_1, C_2, ..., C_m$ contadores, que representa los M elementos usando d funciones de hash tal y como ocurría con los CBF. En cualquier momento, el objetivo del SBF es mantener el tamaño del vector base tan cerca de N bits como sea posible, donde $N = \sum_{j=1..m} \lceil \log C_j \rceil$. Significar que a lo largo del Capítulo asumimos $\log$ como $\log_2$.

Para alcanzar este objetivo, cada contador C_j en la estructura SBF, varía dinámicamente su tamaño de tal forma que tiene el mínimo de bits necesario para contar el número de elementos mapeados en la posición j. Con tal de obtener esta flexibilidad, el espacio ocupado por el vector base incluye $\varepsilon \times m$ extra bits que son estratégicamente colocados entre los contadores. Cada uno de estos bits se añade cada $\lfloor \frac{1}{\varepsilon} \rfloor$ contadores, donde $0 < \varepsilon \leq 1$.

Mientras que el espacio ocupado por el vector base se mantiene cercano al valor óptimo con el fin de alcanzar tal flexibilidad que permita los contadores tener diferentes tamaños, la estructura SBF requiere de complejas estructuras de índices. En el contexto de nuestro Capítulo, identificamos dichas estructuras descritas en [27] como:

- *Coarse Vector* (CV): es un vector de bits que proporciona información del desplazamiento en bits para acceder a un subgrupo de contadores. Los desplazamientos son proveidos usando contadores de un tamaño fijo de bits.
- *Offset Vector* (OV): es un vector de bits que proporciona una información más detallada acerca del desplazamiento de los contadores dentro de cada subgrupo de contadores señalado por CV.

La Figura 7.1 muestra las estructuras de datos utilizadas por un SBF. El primer nivel de coarsificación viene dado por el *Coarse Vector* (CV1), que contiene $\frac{m}{\log N}$ desplazamientos de $\log N$ bits cada uno, de modo que, cada desplazamiento representa un subgrupo de contadores SC. Como se explica en detalle en [27], para el subgrupo de contadores que cumpla $\sum_{C_j \in SC} \log \lceil C_j \rceil < \log^3 N$, un segundo nivel de coarsificación viene dado por el *Coarse Vector* (CV2), proveyendo una información más detallada acerca de los desplazamientos de cada contador $C_j \in$ SC. Por el contrario, en el caso de que un subgrupo de contadores cumpla $\sum_{i \in SC} \log \lceil C_j \rceil \geq \log^3 N$, entonces, como se específica en [27], se utiliza un *Offset Vector* OV, que proporciona el desplazamiento exacto para cada contador. Por simplicidad y sin pérdida de generalidad, los modelos analíticos presentados en este Capítulo asumen el primer caso, en el que se requiere de la presencia del vector CV2. CV2 divide SC en subgrupos (SC$'$) de $\log \log N$ contadores, y mantiene un total de $\frac{\log N}{\log \log N}$ desplazamientos. Puesto que los desplazamientos en CV2 son como máximo $\log^3 N$, cada desplazamiento se puede representar mediante $3 \log \log N$ bits, totalizando $3 \log N$ por cada SC$'$. Finalmente, la información necesitada para localizar la posición exacta del j^{esimo} contador, viene dada por un *Offset Vector* (OV), uno por cada subgrupo SC$'$. El vector OV almacena $\log \log N$ desplazamientos de $3 \log \log N$ bits cada uno, totalizando $3(\log \log N)^2$ bits por subgrupo SC$'$.

El vector OV, puede ser también substituido por una tabla de búsqueda dependiendo de un límite basado en la longitud de SC$'$. Más detalles sobre este caso se pueden encontrar en [27].

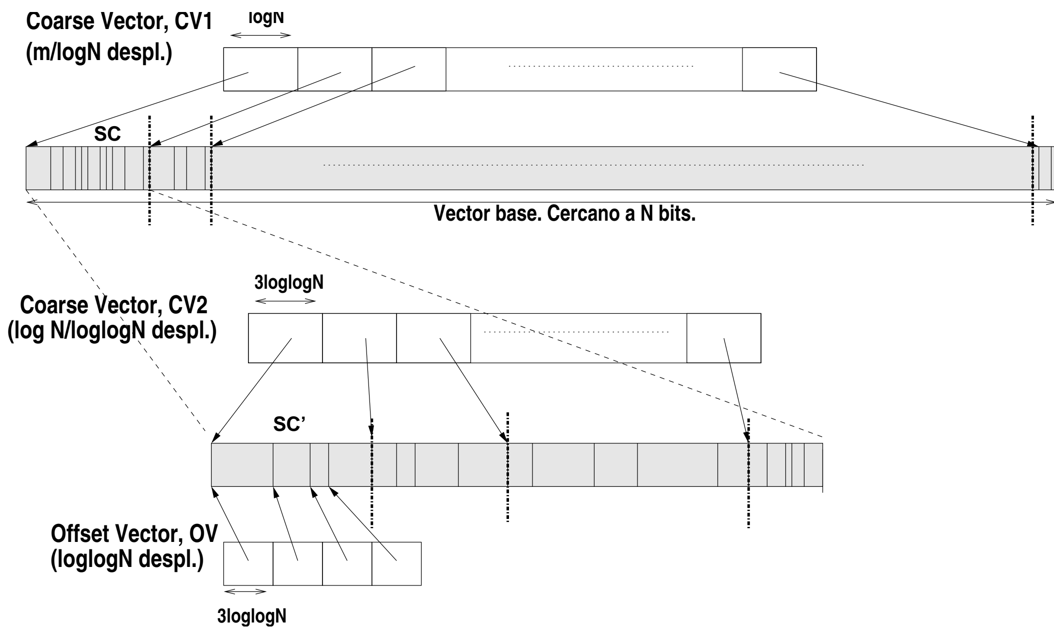


Figura 7.1 Estructuras de datos de los SBF.

7.4 Dynamic Count Filters (DCF)

En esta Sección presentamos los *Dynamic Count Filters* (DCF), que combinan las características de los CBF y los SBF: el rápido acceso de los CBF y la adapatación dinámica de los contadores que presentan los SBF.

7.4.1 La estructura de datos

Un *Dynamic Count Filter* (DCF) está compuesto de dos vectores. El primer vector es, al igual que ocurría con los CBF, un vector de bits en el que cada entrada tiene un tamaño fijo de $x = \log\left(\frac{M}{n}\right)$, donde recordemos que M es el total de número de elementos en el conjunto de datos, y n es el número de valores distintos. Si consideramos que el filtro tiene m contadores (C_j for $j = 1..m$), el vector CBF, que denominamos CBFV, totaliza un tamaño igual a $m \times x$ bits. El segundo vector es el *Overflow Counter Vector* (OFV), que también tiene el mismo número de entradas, cada una incluyendo un contador (OF_j para $j = 1..m$) que mantiene el contaje del número de veces que la correspondiente entrada en el CBFV ha sufrido un *overflow*. El tamaño de cada contador en el OFV cambia dinámicamente dependiendo en la distribución de los elementos en el conjunto de datos representado. Por ejemplo, ante la presencia de sesgo en los datos, el tamaño de un contador será mayor que para distribuciones uniformes. En cualquier momento, el tamaño de cada contador es igual al número de bits requerido para representar el mayor valor almacendo en el OFV ($y = \lfloor \log(\max(OF_j)) \rfloor + 1$). Así pues, el tamaño del OFV totaliza $m \times y$ bits.

La Figura 7.2 muestra las estructuras de datos de las que un DCF está compuesto. A partir de esta Figura es posible observar que la estructura de datos DCF está formada por m entradas, cada una compuesta por un contador consistente en un par de contadores, $\langle OF_1, C_1\rangle$,...,$\langle OF_m, C_m\rangle$. Todos los contadores en el DCF tienen un tamaño igual a $x + y$ bits, donde y varía de forma dinámica su longitud en bits.

La decisión de tener un tamaño fijo para cada contador implica que, por una lado, muchos bits en la estructura de datos no sean utilizados, y por otro lado, que el acceso a ambos vectores sea directo, y por lo tanto, rápido. Así pues, los DCF no realizan un uso óptimo de la memoria, consiguiendo a cambio un rápido acceso a los datos. En conjunto, los contadores de tamaño fijo de la estructura de datos DCF obtienen buenos resultados tanto en rapidez de acceso como en consumo de memoria, ya que proporciona un mecanismo con rápidas operaciones de lectura/escritura, y a la vez ahorra espacio de memoria en un amplio abanico de casos.

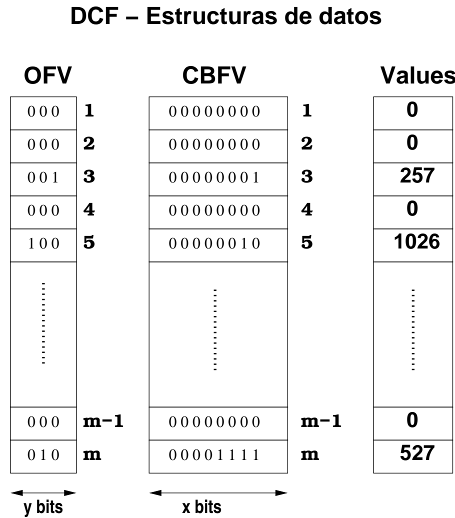


Figura 7.2 La estructura de datos DCF.

7.4.2 Consultar un elemento

Consultar el filtro para un determinado elemento s consiste en chequear d entradas, usando un total de d funciones de hash diferentes, $h_1(s), h_2(s), ..., h_d(s)$. La operación para chequear una entrada del filtro, se realiza accediendo a las entradas correspondientes de los vectores CBFV y OFV. Como los contadores en ambos vectores tienen un tamaño fijo de bits, localizar una entrada es immediato mediante simples operaciones de desplazamiento de bits y módulo. Los bits de cada contador son extraídos usando rápidas operaciones a través de máscaras de bits (AND y OR). Ambos accesos son rápidos con un coste asintótico de $O(1)$. De este modo, para una cierta entrada j, una vez obtenemos C_j y OF_j, el valor compuesto V_j para el contador alamacenado en la posición j de la estructura DCF se calcula como: $V_j = (OF_j \times 2^x + C_j)$.

Así pues, cuando consultamos un elemento s, acabamos obteniendo d valores $V_{h_i(s):i=1..d}$ a un coste de $d \times O(1) \simeq O(1)$. Los valores asociados a s se utilizan dependiendo de la aplicación. Por ejemplo, si se quiere saber la presencia de dicho elemento dentro del conjunto de elementos represenatado, entonces, es necesario verificar que ninguno de los valores $V_{h_i(s):i=1..d}$ sea 0. Si es así, entonces podemos asegurar que s no está en el conjunto de datos representado, de otra forma, s pertenece al conjunto de datos representado con una probabilidad de falso positivo P [14].

7.4.3 Actualizaciones

Cada vez que insertamos o borramos un elemento s, necesitamos actualizar d entradas ($h_1(s)$, $h_2(s)$, ..., $h_d(s)$) en el CBFV y el OFV. Las actualizaciones en el CBFV son rápidas y se realizan en cada contador $C_{h_i(s):i=1..d}$ mediante simples operaciones de incremento (insertado) y decremento (borrado) en una unidad. Las actualizaciones en el OFV son más infrecuentes pero a la vez, pueden resultar más costosas. Además, el OFV puede necesitar aumentar su tamaño cuando se actualizan los contadores $OF_{h_i(s):i=1..d}$ a causa de un *overflow* o *underflow* en el correspondiente contador $C_{h_i(s)}$. En las siguiente Secciones explicamos de forma más detallada los casos de inserción y borrado para un cierto elemento s.

Inserción de un elemento

Como se mencionó anteriormente, cuando se inserta un elemento s, d contadores $C_{h_i(s):i=1..d}$ en el CBFV tienen que ser incrementados en una unidad. En caso de que un contador C_j, para cualquiera de las entradas incrementadas $j = h_i(s) : i = 1..d$, sufra un *overflow* (valor supera $2^x - 1$), entonces, el valor de C_j se inicializa a cero y el correspondiente contador en el OFV, OF_j, tiene que ser incrementado en una unidad. Así pues, la inserción de un elemento requiere como máximo de dos operaciones de lectura y escritura, las cuales tienen un coste asintótico de $O(1)$.

En el caso de que el contador OF_j sufra un *overflow* (supere el valor $2^y - 1$), entonces, antes de poder realizar la operación, un bit tiene que ser añadido a todos los contadores en OFV con el fin de evitar la saturación de contadores. Llamamos a la acción de cambiar el tamaño de OFV, opearción de *rebuild*. Estas operaciones son costosas ya que requieren disponer un nuevo vector, la copia de los contadores del viejo vector al nuevo extendido, y finalmente liberar el espacio ocupado por el antiguo OFV. En consecuencia, como los vectores tienen m entradas, la operación de *rebuild* tiene un coste asintótico de $O(m)$. Significar que aunque la operación de *rebuild* es costosa, la motivación de tener la estructura OFV separada del CBFV reside en que operar sobre el OFV es menos costoso que operar sobre toda la estructura. Esto es debido al hecho de que el OFV es más pequeño que toda la estructura DCF, y también a que muchas de sus entradas tienen una probabilidad más alta de estar a cero, lo cual no sucedería con el CBFV. Significar que el hecho de que un contador sea cero supone que la entrada no necesita ser copiada. En definitiva, aunque el coste asintótico sería el mismo, los movimientos de memoria y el copiado de información para el nuevo vector OFV son menos costosas que si recrearamos toda la estructura DCF.

Borrado de un elemento

Ante el borrado de un elemento s del conjunto de datos representado, d contadores $C_{h_i(s):i=1..d}$ en el CBFV tienen que ser decrementados en una unidad. Cuando uno de los contadores sufre de *underflow* (contador C_j a ser decrementado para cualquiera de las entradas $j = h_i(s) : i = 1..d$, tiene como valor cero), entonces, su valor se inicializa a $2^x - 1$ y su correspondiente contador en el OFV, OF_j se decrementa en una unidad. De esta forma, como pasaba en la operación de inserción, como mucho se ejecutan dos operaciones de lectura y escritura, obteniendo un coste asintótico de $O(1)$. Significar que cuando decrementamos un contador, bien C_j o OF_j deben de ser mayores que cero, ya que no se contempla el borrado de elementos que no han sido previamente insertados.

De forma similar a la operación de *rebuild* causada por la operación de inserción, la operación de borrado también puede resultar en una operación de *rebuild*. En este caso, sin embargo, con el objetivo de liberar espacio de memoria. En el momento en que un contador OF_j decrementa su valor de 2^{y-1} a $2^{y-1} - 1$ debemos de chequear todos los contadores del OFV, de forma que

si todos sus valores son menores que 2^{y-1}, entonces, podemos disminuir el tamaño de OFV en un bit por contador. Mientras que en la teoría esta operación necesita chequear todos los valores de los contadores, a la práctica esta operación es simple si tenemos una pequeña estructura con contadores que mantenga la información de los bits utilizados por los m contadores. Detallamos esta optimización en la siguiente Sección. Aumentar o disminuir el tamaño del OFV resulta en la misma operación de *rebuild* de la estructura DCF, que tiene un coste asintótico de $O(m)$.

7.4.4 Control de la operación de rebuild

La principal diferencia entre la operación de *rebuild* por inserción y la operación de *rebuild* por borrado es que, mientras que la primera se realiza para evitar la saturación de contadores, la segunda es opcional y puede ser retrasada con el fin de evitar situaciones inestables como consecuencia de operaciones de inserción y borrado que pudieran causar un excesivo número de operaciones de *rebuild*.

En consecuencia, introducimos un límite entre los valores 2^{x+y-2} y $2^{x+y-1} - 1$. Definimos dicho límite como $T = 2^{x+y-2} + (2^{x+y-1} - 2^{x+y-2}) \times \lambda$, donde λ toma valores entre 0.0 y 1.0. De este modo, cuando decrementamos en una unidad una entrada j del DCF, y siendo V_j el valor asociado al contador $\langle OF_j, C_j \rangle$, entonces, si $V_j < T$, realizamos la operación de *rebuild* cuando todos los contadores en el DCF cumplan: $\forall j : j = 1..m : V_j < T$.

Mantenimiento del límite

Como se explicó en la Sección 7.4.3, con el fin de evitar chequear el valor de todos los contadores para detectar *underflow*, realizamos una optimización y mantenemos una estructura de contadores de *overflow*. Llamamos l al nivel de *overflow* de cualquier contador en la estructura DCF. El nivel de *overflow* representa el número de bits usados por el contador de *overflow* OF, así pues, l puede tomar valores entre 0 e y. Consecuentemente, una entrada j en el DCF tiene un nivel de *overflow* $l > 0$ si su contador de *overflow* OF_j tiene un valor $2^{l-1} < OF_j \leq 2^l - 1$. Por otro lado, tienen un nivel de *overflow* $l = 0$ si $OF_j = 0$. Mantenemos los diferentes contadores de niveles de *overflow* en una estructura llamada *Contador de niveles* (CL).

Con el objetivo de utilizar el límite T tal y como hemos descrito anteriormente, mantenemos dos contadores por nivel en CL: (1) SM, un contador para el número de contadores OF_j cuyo valor es menor o igual que el límite para el nivel al que pertenecen T_l,, *i.e.* contadores por debajo de $2^{x+l-2} + (2^{x+l-1} - 2^{x+l-2}) \times \lambda$ para valores con un nivel mayor que 0, y por debajo de $(2^x - 1) \times \lambda$ para contadores con nivel 0; y (2) LG, un contador para el número de contadores OF_j con un valor igual o mayor que el límite T_l. El nivel l de un contador almacenado en la posición j en la estructura DCF, se calcula como:

$$\begin{cases} 0 & \text{if } OF_j = 0; \\ \lfloor \log(OF_j) \rfloor + 1 & \text{cualquier otro caso.} \end{cases}$$

Entonces, cuando un elemento s es borrado o insertado, y los valores $V_{h_i(s):i=1..d}$ incrementados o decrementados, debemos actualizar los correspondientes contadores de nivel.

Significar que la memoria necesitada por la estructura CL es negligible: sólo tenemos $2 \times (y+1)$ contadores. También, el proceso de actualización tiene un coste asintótico $O(1)$, ya que sólo se ejecuta una simple operación de suma, resta, y comparación.

La Figura 7.3 muestra un ejemplo de como la estructura CL se utiliza para retrasar las operaciones de *rebuild* del OFV causados por operaciones de borrado de elementos. En este ejemplo mostramos

una estructura DCF con 8 contadores, $m = 8$, y $\lambda = 0.5$. El CBFV tiene $x = 4$ bits por contador, y después de varias inserciones el OFV tiene $y = 2$ bits. El valor decimal de cada contador se muestra sólo para una mayor claridad del ejemplo ya que tienen el mismo valor que sus correspondientes par de contadores. Inicialmente, en la parte izquierda, los valores de cada contador son $\{0, 2, 7, 31, 9, 28, 17, 60\}$, de modo que, los contadores en las posiciones 0, 1, 2, y 4 tienen un nivel $l = 0$ (con tres contadores por debajo del límite T_0 y dos por encima), y el último contador tiene un nivel $l = 2$ (con el contador por encima de T_2). En la parte central de la Figura 7.3, podemos observar que, después de múltiples operaciones de borrado, no hay contadores con nivel 2 y nivel 1 por encima del límite T_2. Así pues, como todos los contadores en el DCF están por debajo del límite T_2, se puede realizar el *rebuild* del OFV eliminando el bit de más peso de cada contador, y consecuentemente reduciendo el tamaño de la estructura DCF.

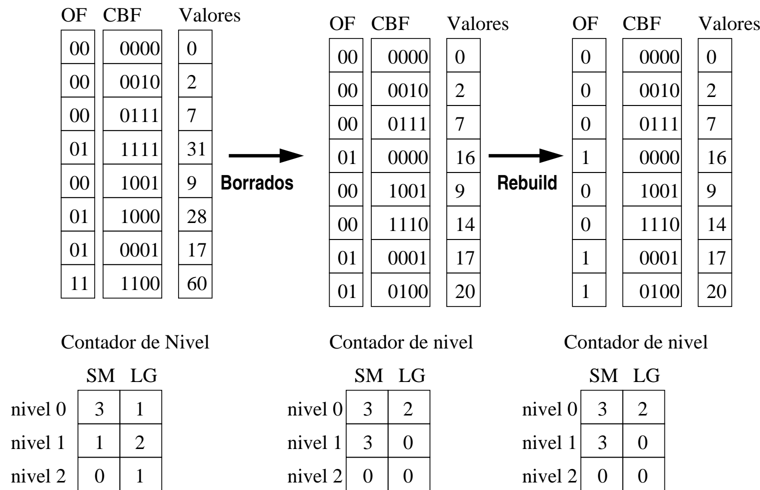


Figura 7.3 Contador de niveles.

Valor óptimo del límite T, λ

Las operaciones de *rebuild* son las más costosas en la estructura DCF. Por esta razón, definimos como límite óptimo para DCF, el valor de λ que minimiza el número de operaciones *rebuilds* a lo largo del tiempo de ejecución.

Primero de todo, debemos determinar cuales son las situaciones bajo las cuales merece la pena realizar un *rebuild* de la estructura en caso de borrado. Para esto, definimos el *ratio* $R = n_{inserts}/n_{deletes}$, el cual es una métrica que nos indica como el número de operaciones de inserción evoluciona comparado con el número de operaciones de borrado a lo largo del tiempo.

La Figura 7.4 muestra la evolución del DCF en términos del número de *rebuilds* ejecutados a lo largo del tiempo de ejecución. La gráfica incluye tres lineas, representando tres escenarios diferentes, T1, T2, y T3, cada uno con un *ratio* diferente: T1 para $R > 1$, T2 para $R \simeq 1$, y T3 para $R < 1$.

Los resultados en esta gráfica representan, en media, el número de operaciones de *rebuild* para 10 ejecuciones utilizando diferentes valores de λ, que pueden ir de 0.0 a 1.0. El primer paso para todos los escentarios consiste en operaciones de inserción para los M elementos del conjunto que se quiere representar. Después de haber insertado todos los elementos, cada paso en el tiempo se

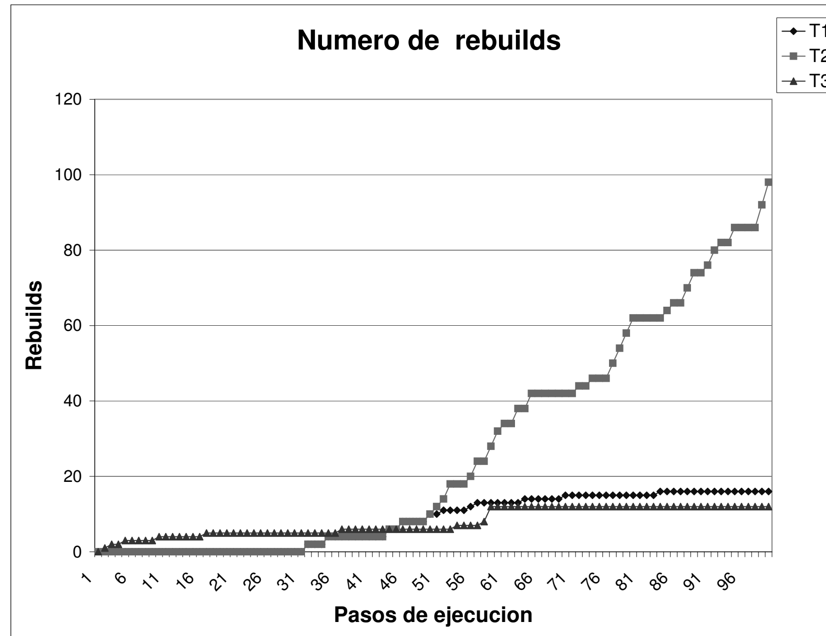


Figura 7.4 Número de operaciones de *rebuild* del DCF para los tres distintos escenarios: T1 ($\frac{n_{inserts}}{n_{deletes}} > 1$), T2 ($\frac{n_{inserts}}{n_{deletes}} \simeq 1$), and T3 ($\frac{n_{inserts}}{n_{deletes}} < 1$).

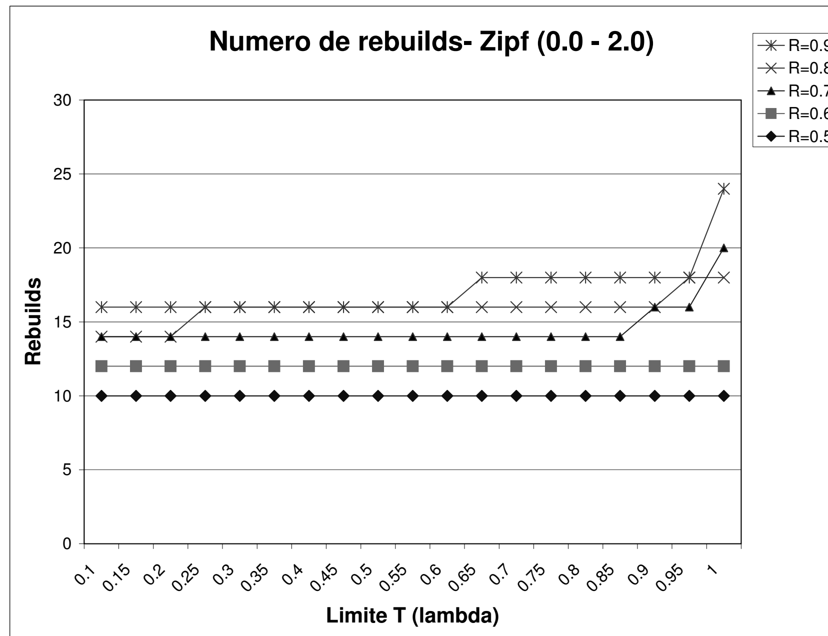
compone de varias operaciones de inserción y borrado. Las inserciones de elementos siguen un distribución Zipfian [18], en la que el sesgo de los datos viene dado por la variable θ, cuyo valor oscila entre 0.0 y 2.0, y es escogido de forma aleatoria en cada paso de ejecución. Los valores insertados, son a la vez aleatoriamente escogidos para las operaciones de borrado.

A partir de la Figura 7.4 podemos identificar el siguiente comportamiento:

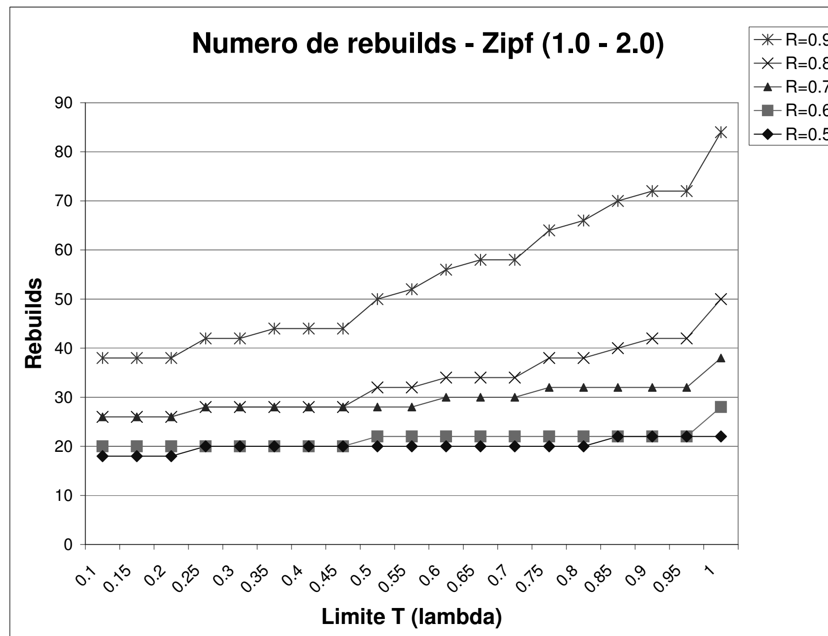
- T1 muestra en media para cualquier λ, el número de operaciones de *rebuild* ejecutadas cuando R crece a lo largo del tiempo. En este caso, siempre incrementamos el número de operaciones de *rebuild* innecesarias causadas por operaciones de borrado, por ejemplo, en un cierto período corto de tiempo, podemos tener un *rebuild* causado por una inserción, por el simple hecho que las inserciones son más frecuentes.

- T2 y T3 muestran que, en media para cualquier λ, el número de operaciones de *rebuild* no se incrementa cuando $R \simeq 1$ y $R < 1$. De este modo, para los escenarios T2 y T3 puede existir un valor de λ que minimiza el número de operaciones de *rebuild*, y que mejora el uso de la memoria por parte de los DCF.

La Figura 7.5-(a) muestra para diferentes valores de $R < 1$, cuales son los valores para λ que minimizan el número de operaciones de *rebuild*. Ambas gráficas demuestran que para *ratios* $R \leq 0.6$, en los que el número de elementos borrados es mucho mayor que el número elementos insertados, independientemente del valor de λ que escojamos, siempre tenemos un número constante de *rebuilds*. Sin embargo, para $R > 0.6$, el valor establecido para λ es importante, y valores para $\lambda \simeq (1 - R)$ son aquellos que minimizan el número de *rebuilds*. La Figura 7.5-(b) muestra este hecho con mayor claridad. Cuando los datos que insertamos tienen una distribución con mucho



(a)



(b)

Figura 7.5 Número de operaciones de *rebuild* en el DCF para diferentes valores de λ, y para diferentes *ratios* que siguen una distribución Zipfian con θ entre (a) 0-2 y (b) 1-2. R es el *ratio* $\frac{n_{inserts}}{n_{deletes}}$ utilizado para la la ejecución dada.

sesgo, es decir, las inserciones siempre afectan a los mismos contadores, entonces, el número de *rebuilds* es más sensible al valor escogido para λ. Podemos observar que para $\lambda = (1-R)$ tenemos el menor número de operaciones de *rebuild*. Dada esta conclusión, la aplicación podría ser usada como un tipo de predictor basado en los hechos observados a lo largo del tiempo de ejecución, y de este modo, cambiar el valor de λ si fuera necesario. Para nuestro trabajo hemos asumido una distribución 50-50 de operaciones de inserción y borrado, $\lambda = 0.5$.

7.5 DCF versión particionada (PDCF)

La principal desventaja del DCF reside en el hecho de que añade un bit a todas las entradas del OFV durante la operación de *rebuild*. Este hecho representa un desperdicio de memoria ya que puede crear mucho espacio desaprovechado por los contadores, especialmente para distribuciones con mucho sesgo en los datos. Además, este hecho hace que el coste de la operación de *rebuild* sea alto porque es necesario actualizar todas las entradas del OFV.

A continuación presentamos *Partitioned Dynamic Count Filters* (PDCF), que consiste en una optimización de los DCF que intenta solventar los problemas mencionados.

7.5.1 La estructura de datos

PDCF divide DCF en múltiples particiones tal que, sólo aquellas particiones en las que se produce un *overflow*/*underflow* son objeto de la operación de *rebuild*. El objetivo reside en mejorar el uso de la memoria reduciendo el espacio no utilizado por los contadores, y al mismo tiempo reduciendo el coste de la operación de *rebuild* ya que sólo se ha de actualizar la partición afectada. En consecuencia, es de esperar que se mejore el *throughput* y la calidad de servicio. Las propiedades ventajosas de PDCF se ilustran mejor en la Figura 7.6. Es de importancia comparar los requerimientos de espacio para los mismos valores en los contadores según las diferentes estruturas, DCF, PDCF con 2, 4, y 6 particiones.

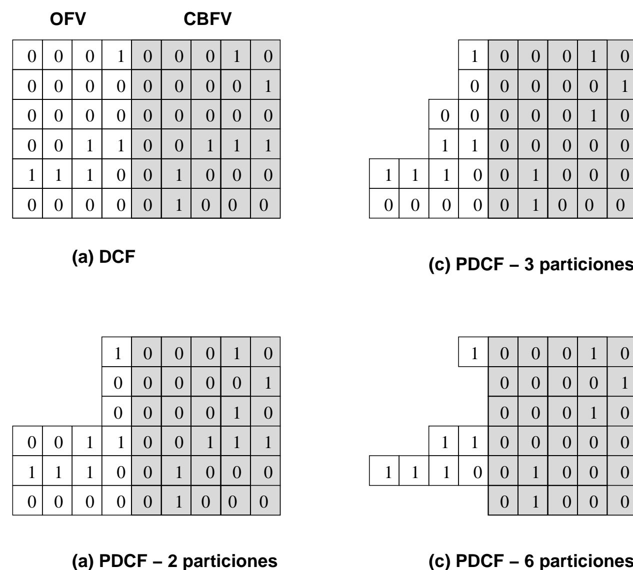


Figura 7.6 *Partitioned Dynamic Count Filters* (PDCF). La idea conceptual.

Las estructuras requeridas para la implementación de un PDCF se muestran en la Figura 7.7. Aunque los PDCF se basan en las estructuras básicas presentadas en los DCF, el enfoque particionado introduce un cúmulo de modificaciones a los ya existentes vectores, además, requiere de una nueva estructura para gestionar las particiones. Un PDCF está formado por las siguientes estructuras:

- Chun**K** **C**ount **B**loom **F**ilter **V**ector (CBFV$_K$). CBFV$_K$ es la estructura de datos utilizada para representar la K^{esima} partición del CBFV presentado para los DCF. Las m entradas del CBFV se particionan de forma lógica en γ particiones ($K = 1..\gamma$), de modo que, cada partición está representada por un CBFV$_K$ (CBFV$_1$, CBFV$_2$,..., CBFV$_\gamma$). Todo CBFV$_K$ tiene el mismo número de entradas, $m' = \lceil \frac{m}{\gamma} \rceil$, y consiste en un contador con un tamaño fijo de bits: contadores CK_1,CK_2...,$CK_{m'}$ de $x = \left\lceil \log\left(\frac{M}{n}\right) \right\rceil$ bits cada uno. Así pues, cada partición tiene un total de $m' \times x$ bits, y toda la estructura tiene el mismo número de bits que el CBFV de DCF, $\gamma m' \times x \simeq m \times x$ bits.

- Chun**K** **O**ver**F**low **V**ector (OFV$_K$). OFV$_K$ está formado por m' entradas (contadores OK_1,OK_2,...,$OK_{m'}$), dónde cada entrada, al igual que ocurría en el OFV para los DCF, cuenta el número de *overflows* sufridos por el contador en el correspondiente CBFV$_K$. Hay una estructura OFV$_K$ por cada CBFV$_K$, (OFV$_1$, OFV$_2$,..., OFV$_\gamma$). Todos los contadores que pertenecen a un OFV$_K$ tienen el mismo tamaño en bits. Sin embargo, el tamaño de estos contadores puede cambiar de forma dinámica en el tiempo dependiendo de la distribución de los valores en el conjunto de datos representado. Para cada OFV$_K$, el número de bits por contador es igual al número de bits requerido para representar el mayor valor almacenado en esa partición $y_K = \lfloor \log(\max(OK_j : j = 1..m')) \rfloor + 1$, asumiendo que al menos hay un bit por contador. De esta forma, el tamaño de cada OFV$_K$ totaliza $m' \times y_K$ bits.

- **P**artitioning **V**ector (PV). PV es el vector de particionado y tiene un total de γ entradas que conectan cada CBFV$_K$ con su correspondiente vector de *overflow* OFV$_K$. Cada entrada en PV almacena dos campos: (i) un puntero al la estructura OFV$_K$ y, (ii) un contador que determine el valor de y_K. Significar que el tamaño del contador y_K puede diferir entre particiones, lo cual implica mantener un contador para cada partición con el fin de determinar su tamaño en bits.

Para una mejor comprensión, la Figura 7.8 muestra un visión general de la propuesta particionada PDCF. Un PDCF es un DCF γ-particionado, donde cada partición consite en un par de vectores , $\langle OFV_1, CBFV_1 \rangle$, $\langle OFV_2, CBFV_2 \rangle$, ... ,$\langle OFV_\gamma$, CBFV$_\gamma \rangle$. Cada par de vectores $\langle OFV_K, CBFV_K \rangle$ está formado por m' entradas ($m' = 5$ en la Figura 7.8), cada una representando un contador dividido en dos contadores de bits $\langle OK_1, CK_1 \rangle$, $\langle OK_2$, $CK_2 \rangle$,..., $\langle OK_{m'}, CK_{m'} \rangle$. Dentro de cada partición $\langle OFV_K, CBFV_K \rangle$, todos los contadores tienen el mismo tamaño: $x+y_K$ bits. Sin embargo, contadores de diferentes particiones pueden diferir en tamaño porque el valor y_K es independiente para cada OFV$_K$.

En definitiva, los PDCF están destinados a realizar un mejor uso de la memoria mediante la segmentación de contadores en diferentes grupos. Como tal, no todos los contadores crecen a causa de un sólo *overflow*/*underflow*, y las operaciones de *rebuild* resultan más eficientes para PDCF en conparación con DCF. Significar que el particionado no empeora el tiempo de acceso a la estructura de datos.

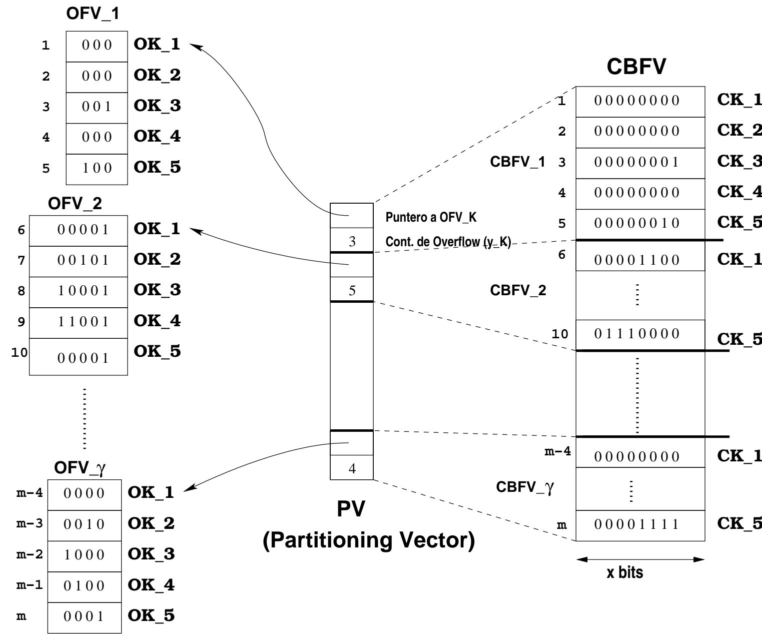


Figura 7.7 PDCF. La estructura de datos.

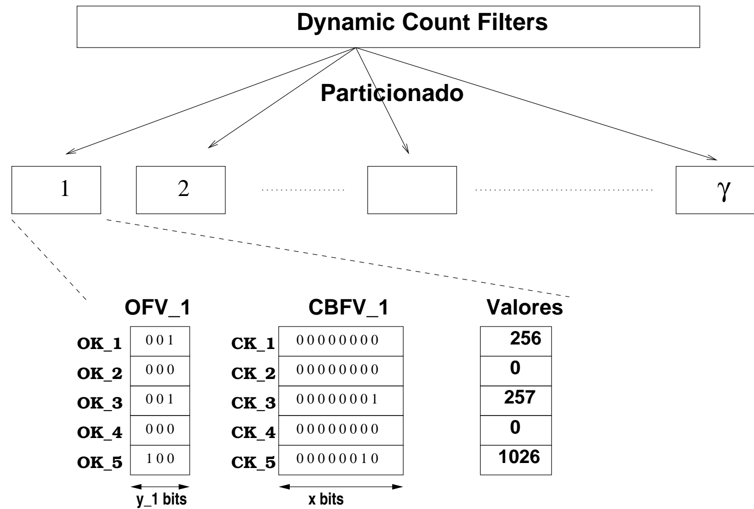


Figura 7.8 Visión de los PDCF.

7.5.2 Actualización, consulta, y operación rebuild

Todas las operaciones de inserción, borrado, consulta y *rebuild* de la estructura PDCF son iguales que las explicadas para los DCF pero aplicadas a la partición que corresponda. Cuando se actualice o consulte un elemento, los valores obtenidos por las d funicones de hash, $h_1(s), h_2(s), ..., h_d(s)$, mapearán contadores correspondientes a diferentes particiones sobre las cuales se ejecutará la respectiva operación. Las operaciones añadidas respecto a los DCF aplican de la siguiente forma;

(i) determinar la partición K correspondiente a cada valor de las d funciones de hash. Para ello, realizamos las divisiones $K(i) = \frac{h_i(s)}{m'}$ para $i = 1..d$, cada división con un coste asintótico de $O(1)$; (ii) determinar la entrada j para cada partición Kmediante una operación de módulo, $j = h_i(s) \bmod m'$, que también tiene un coste asintótico de $O(1)$. Significar que $m' = \lceil \frac{m}{\gamma} \rceil$ es un valor precalculado; (iii) consultar la entrada K del PV con el fin de averiguar la dirección del correspondiente del OFV$_K$ y el tamaño en bits de los contadores asociados y_K.

Como se ha dicho, la operación de *rebuild* ya sea por inserción o borrado, también es igual que la explicada para los DCF. En este caso, al aplicarse a una partición, su coste asintótico es de $O(m')$. También, como operación adicional y de coste despreciable, debe de actualizarse el contador para y_K de la correspondiente entrada en PV.

7.5.3 Número óptimo de particiones (γ)

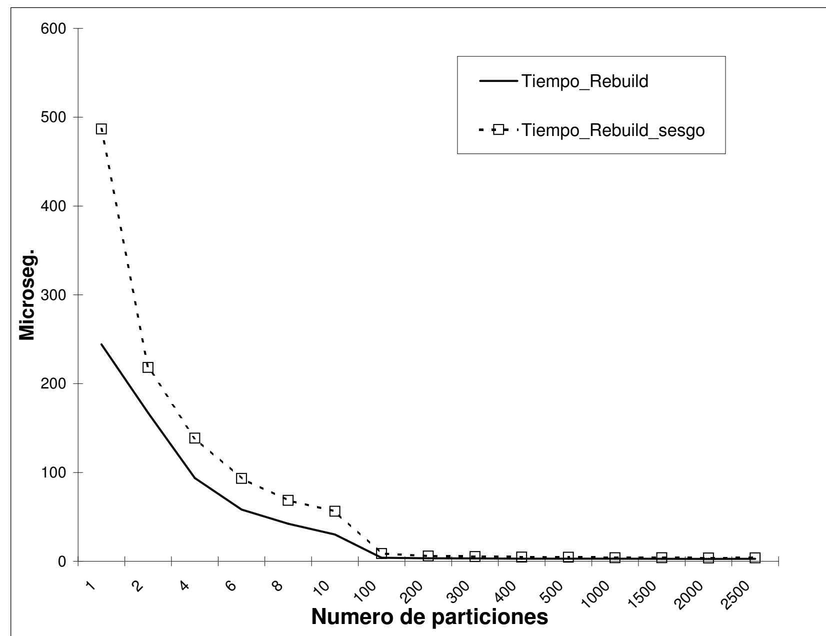
Definimos el número óptimo de particiones en función de la memoria ocupada, y del tiempo de ejecución para la operación de *rebuild* de un OFV$_K$. Durante la operación de *rebuild*, debemos realizar el *lock* de la estructura OFV$_K$, de modo que, durante el tiempo que dure la operación no se pueden realizar actualizaciones ni consultas, y por lo tanto se necesita de espacio adicional de memoria para mantener todas las peticiones recibidas durante dicha operación. Denominamos a este espacio adicional, *buffer area*.

A modo de ejemplo, la Figura 7.9 muestra, para diferentes números de particiones, los resultados, del tiempo medio de ejecución de la operación de *rebuild*, y la media de la memoria utilizada para $n = 1000$ valores distintos y un total de $M = 100000$ elementos. Para esta configuración, los elementos son primero insertados, depués consultados, y finalmente borrados. Presentamos resultados tanto para distribuciones uniformes como no uniformes (usando una distribución Zipfian [18]), de este modo, podemos tener una idea intuitiva de lo que pasaría con distribuciones intermedias.

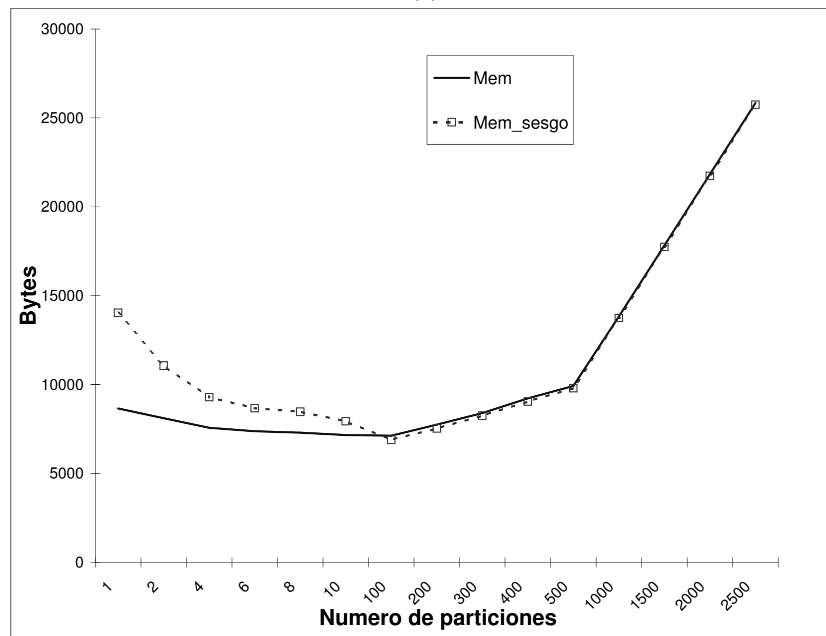
La Figura 7.9-(a) muestra que el tiempo de ejecución de la operación de *rebuild* se incrementa a medida que el número de particiones aumenta. El tiempo de *rebuild* se reduce en un orden de magnitud comparado con el caso de una sola partición (equivalente a DCF). Los PDCF particionan la zona de *overflow* de los DCF, de tal forma que salvan espacio de memoria para los contadores, y reducen el tiempo de ejecución de la operación de *rebuild*.

Los resultados relativos a la memoria se muestran en la Figura 7.9-(b). En ella, podemos observar que los mejores resultados se van a producir para distribuciones no uniformes, en las que se reduce hasta el doble la memoria utilizada si se compara con el caso de una sola partición (DCF). También podemos observar que las ganancias en memoria están limitadas por el número de particiones. Por encima de 200 particiones se observa un crecimiento de los requerimientos de memoria respecto al caso de una particicón. Esto es debido a que el tamaño del PV empieza a ser mayor que el ahorro de memoria obtenido por la reducción en el espacio utilizado por los contadores y una *buffer area* más pequeña. Sin embargo, significar que llegado este punto, un mayor número de particiones causa efectos despreciables en el tiempo de *rebuild*. Así pues, para el caso testeado, el número óptimo estaría por debajo de 200 particiones. En los siguientes parráfos nos disponemos a analizar este hecho.

Una tendencia genérica para cualquier número de valores distintos n y número total de elementos M, es que a medida que aumentemos el número de particiones, observamos que (i) el tiempo de *rebuild* disminuye, reduciendo a su vez la *buffer area*, y (ii) el tamaño de PV necesitado para gestionar las particiones, incrementa. A continuación estudiaremos la forma de obtener el número



(a)



(b)

Figura 7.9 (a)Tiempo de ejecución de la operación de *rebuild* ; (b) Memoria utilizada. Resultados para distribuciones uniformes y distribuciones no uniformes.

óptimo de particiones γ que permita reducir el tiempo de *rebuild* y a la vez consumir menos memoria.

Para el siguiente análisis asumimos una distribución uniforme de los datos, y una perfecta distribución de las funciones de hash. También utilizamos las siguientes definiciones:

- γ número de particiones.
- t_R tiempo de *rebuild* para una sola partición ($\gamma = 1$, DCF).
- S_p tamaño de un puntero en bytes.
- S_c tamaño de un contador en bytes.
- S_e tamaño de un elemento en bytes.
- $freq$ frecuencia de las peticiones entrantes, $\frac{requests}{second}$.

Definimos $fmem$ como una función para estimar el ahorro de memoria dependiendo del número de particiones γ. Definimos $fmem$ de la siguiente forma:

$$fmem = t_R \left(1 - \frac{1}{\gamma}\right) freqS_e - (S_p + S_c)\gamma$$

donde $t_R(1 - \frac{1}{\gamma})freqS_e$ es el ahorro de memoria estimado debido al tamaño de la *buffer area* utilizado para las peticiones entrantes. En otras palabras, $t_R freqS_e$ es el espacio necesitado por PDCF para una sóla partición, y $\frac{t_R}{\gamma} freqS_e$ es el tamaño de la *buffer area* cuando el número de particiones es $\gamma > 1$. $(S_p + S_c)\gamma$ es el espacio adicional necesitado para mantener las γ particiones. Significar que el ahorro en memoria para datos con sesgo en los PDCF es mayor que en los DCF debido al hecho que se expanden menos particiones. Sin embargo, como asumimos una distribución uniforme de los datos, el número de particiones expandidas totaliza el mismo tamaño en memoria que el vector OFV de los DCF.

Así pues, el valor de γ que maximiza $fmem$ viene dado por $fmem'(\gamma) = 0$. Primero calculamos:

$$fmem'(\gamma) = \frac{t_R freqS_e}{\gamma^2} - \gamma(S_p + S_c)$$

entonces, para $fmem'(\gamma) = 0$, podemos aislar γ de tal modo que:

$$\gamma = \sqrt{\frac{t_R freqS_e}{S_p + S_c}} \tag{7.1}$$

Esta ecuación nos indica que γ es proporcional al tiempo de *rebuild*, la frecuencia $freq$, y al tamaño de los elementos del conjunto represenatdo S_e. El tiempo de *rebuild*, como se confirmará a través de los resultados, se prevee que esté fuertemente influenciado por el número de valores distintos n. Por otro lado, S_e y $freq$ dependerán del contexto de la aplicación. La ecuación indica que cuanto mayores son los valores de estos tres parámetros, mayor será la mejora en términos de *throughput*. Esto se podía esperar ya que cuanto mayores son S_e y $freq$, mayor es el ahorro en consumo de memoria debido a la reducción del tamaño de la *buffer area* ($\frac{t_R}{\gamma} freqS_e$), permitiendo así un mayor número de entradas en el PV.

En la práctica, S_e, S_p y S_c son valores conocidos. Los valores para t_R y $freq$ pueden ser bien, estimaciones o valores observados en ejecuciones previas.

7.6 Comparativa entre SBF,DCF y PDCF

La mayor diferencia entre los SBF y los DCF reside en que la primera tiene como objetivo construir un vector base compacto de tamaño tan cercano a N como sea posible, mientras que la estructura DCF deja perder algo de espacio en el vector base a cambio de un acceso más eficiente. Por otro lado, los PDCF heredan las mejores cualidades de los SBF y los DCF, y reducen el tamaño del area de *overflow* de los DCF, reduciendo la memoria utilizada por los contadores y el tiempo de *rebuild*.

Para los siguientes análisis, usamos la misma terminología que la utilizada en la Sección 7.3.

7.6.1 Recursos de memoria

Realizamos un análisis que modela las diferencias entre los tres enfoques desde el punto de vista del uso que realizan de la memoria. Significar que no se tiene en cuenta el tamaño de la *buffer area* definido en la Sección 7.5.3. También, los modelos que presentamos asumen que los datos, en caso de presentar sesgo, lo hacen por un único valor. De modo que, el procentage de sesgo sk, significa que el sk% de los datos se mapean sobre los mismo contadores, y que el $(100 - sk)$% de los elementos restantes están uniformemente distribuidos.

Basado en el análisis de la estructura SBF presentada en [27] y en la Sección 7.3 de este Capítulo, es posible determinar que los SBF necesitan m bits para CV1, $3m$ bits para CV2, y $3m \times \log\log N$ bits para el OV. Así pues, concluimos que la memoria utilizada por los SBF es: $Mem_{SBF} = m + 3m + 3m \times \log\log N$, donde $N = m + \varepsilon \times m$. Respecto a los DCF, la memoria utilizada se calcula como: $Mem_{DCF} = m \times (x + y)$, donde $\log\left(\frac{M}{n}\right) \leq (x + y) \leq \log(M)$.

La Figura 7.10-(a) muestra como el *ratio* $\frac{Mem_{SBF}}{Mem_{DCF}}$ evoluciona dependiendo del sesgo en los datos y del número total de elementos M. Siendo n el número de valores distintos, M varía proporcionalmente desde 10 a 100000 valores repetidos por cada valor distinto en los elementos. El peor de los escenarios para la estructura DCF (*ratio* inferior a 1), sería aquel en que todos los elementos son mapeados en los mismos contadores (100% de sesgo en los datos). De esta forma, $(x+y)$ alcanza $\log(M)$ en algún momento del tiempo de vida de la aplicación. También, Mem_{DCF} podría se mayor que Mem_{SBF} cuando el número de valores repetidos es extremadamente alto, $100000 \times n$. En todos los demás casos, DCF necesita menos memoria que SBF, especialmente para distribuciones uniformes (0%-20% de sesgo en los datos).

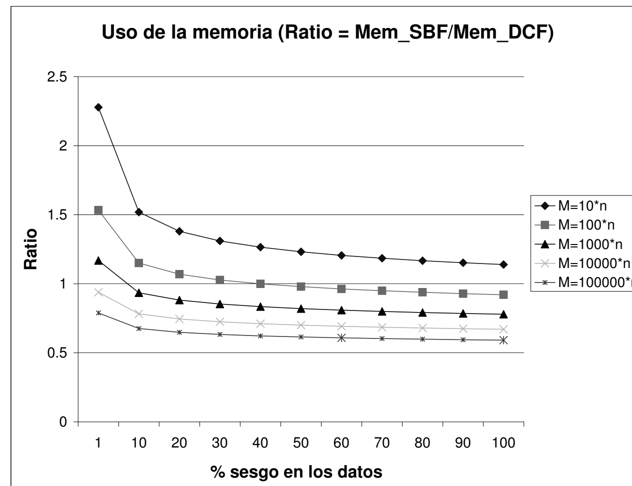
Con el fin de comparar los PDCF con los DCF y los SBF, definimos la ocupación de memoria de los PDCF en función de Mem_{DCF}:

$$Mem_{PDCF} = Mem_{DCF} - \left[\frac{m}{\gamma} y \times (1 - \gamma)\frac{sk}{100}\right] + 6\gamma.$$

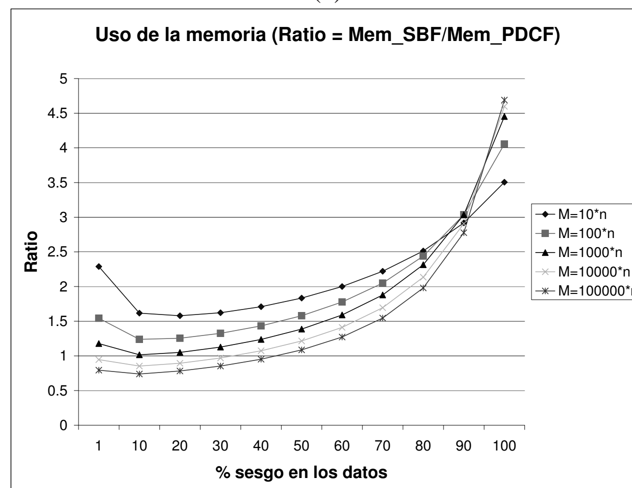
donde $\frac{m}{\gamma} y \times (1 - \gamma)\frac{sk}{100}$ es la cantidad de memoria, en bytes, ahorrada por las particiones no expandidas, y $6 \times \gamma$ es la memoria utilizada por el PV asumiendo que los punteros son 4 bytes y los contadores son 2 bytes. Significar que asumimos un valor fijo para el tamaño y la frecuencia de los elementos. De este modo, aplicando la ecuación 7.1, y asumiendo que t_R es proporcional a n, entonces, γ es también proporcional a la raiz cuadrada de n.

La Figura 7.10-(b) muestra el *ratio* $\frac{Mem_{SBF}}{Mem_{PDCF}}$. En la gráfica correspondiente podemos observar que:

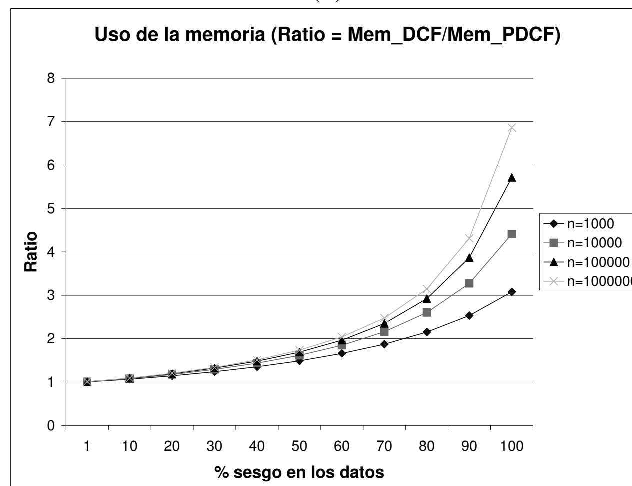
- Los PDCF siempre superan a los SBF (*ratio* por encima de 1) excepto para aquellos casos extremos en los que podemos tener un alto número de valores repetidos con una distribución uniforme.



(a)



(b)



(c)

Figura 7.10 Uso de la memoria (a) SBF contra DCF; (b) SBF contra PDCF; (c) DCF contra PDCF.

- Los PDCF compensan los malos escenarios para los DCF mostrados en la Figura 7.10-(a). En esa gráfica hemos podido observar que los DCF tiene un mal comportamiento cuando hay sesgo en los datos. En estos casos, los PDCF sólo expanden unas pocas particiones, lo que resulta en un ahorro de memoria que se acentúa a medida que el sesgo en los datos es mayor.

Confirmamos estas observaciones a partir de la Figura 7.10-(c) donde mostramos el *ratio* $\frac{Mem_{DCF}}{Mem_{PDCF}}$. Podemos observar que (i) el *ratio* está siempre por encima de 1, significando que los PDCF siempre mejoran el uso de la memoria comparado con los DCF, y (ii) que cuanto mayor es el sesgo en los datos y el número de elementos, mayor es el ahorro en recursos de memoria. Podemos observar este hecho en la Figura 7.9-(b) para valores de $\gamma < 200$.

7.6.2 Tiempo de acceso para localizar un contador

Acceder a un contador en la estructura SBF significa: (i) acceder a CV1 para calcular el desplazamiento inicial para el subgrupo SC en el que C_j está localizado, con un coste de $O(1)$; (ii) acceder a CV2 para calcular el inicio del subgrupo SC' dentro de SC, con coste $O(1)$; y (iii) calcular el desplazamiento exacto para C_j iterando a través de OV, con un coste de $O(\log\log N)$. Así pues, tenemos un coste acumulado de: $Lookup_{SBF} = O(\log\log N)$.

De otra forma, los DCF no necesitan usar estructuras índice para localizar a los contadores. Esto es debido al hecho que todos los contadores tienen la misma longitud en bits. Entonces, acceder al j^{esimo} contador en los DCF es inmediato y tiene un coste de: $Lookup_{DCF} = O(1)$.

Con respecto a los DCF, los PDCF sólo añaden una operación de división junto a una operación de módulo para localizar la partición, y su correspondiente entrada para el valor dado respectivamente. El coste de estas dos operaciones puede ser considerado despreciable en el tiempo total de acceso y podemos concluir que el coste asintótico es $Lookup_{PDCF} = Lookup_{DCF}$.

7.6.3 Operaciones de actualización

Para los SBF, DCF, y PDCF, cada vez que insertamos o borramos un elemento s en el filtro, debemos de actualizar los contadores indicados por los valores devueltos por las funciones de hash, $h_1(s)$, $h_2(s)$,..., $h_d(s)$. Entonces, para los SBF, tenemos que localizar la posición para el contador en el vector base, lo cual tiene un coste de $Lookup_{SBF} \times d$. Para los DCF y PDCF el acceso a los contadores es directo y tiene un coste de$Lookup_{DCF} \times d$. Una vez localizamos los contadores, se ejecutan simples operaciones de incremento o decremento del contador en una unidad.

Teóricamente, la diferencia entre los costes asintóticos no puede parecer relevante, sin embargo, y como se demostrará en la Sección de resultados, el coste computacional añadido por SBF para acceder a las estructuras índices es de importancia en comparación al acceso directo que presentan los DCF y PDCF, repercutiendo de forma directa en el tiempo de actualización.

7.6.4 Rebuilds de la estructura

La inserción de un elemento en la estructura SBF puede resultar en un incremento del tamaño de los contadores en el vector base. En los SBF, el tamaño de un contador se incrementa añadiendo el primero de los bits libres adicionales disponibles. Estos bits adicionales están situados enre los contadores pero no tienen porque estar adyacentes a los contadores que los requieren. A causa de esto, cuando un contador necesita incrementar su tamaño, tiene que buscar el bit libre disponible más cercano, y después desplazar todos los contadores de entre medio con el fin de hacer el hueco requerido por el contador. Así pues, el coste de una operación de *rebuild* depende de lo cerca que

esté el primer bit libre adicional, y es lineal respecto al número de contadores afectados en el vector base. En [27], se realizan suposiciones acerca del número máximo de inserciones con el fin de dar un coste asintótico a la operación de *rebuild*. En este trabajo, con el objetivo de proveer un análisis cuantitavio del coste de la operación de *rebuild*, asumimos total aleatoriedad en los datos entrantes a la aplicación. Además, en las siguientes Secciones mostraremos resultados reales de tiempo para la operación de *rebuild*.

Siendo C_j el contador que necesita ser expandido, y C_l el contador más cercano con el bit libre a su lado, entonces, el número de posiciones envueltas en la operación son $l - j + 1$ si $l \geq j$, o $j - l$ en cualquier otro caso. Así pues, para cada posición debemos de localizar el contador en la posición que le corresponde ($Lookup_{SBF}$), y copiar cada contador con el nuevo desplazamiento asociado al vector auxiliar. El tiempo requerido para localizar un contador añade complejidad a la operación de *rebuild*, y en el peor de los casos, el coste asintótico es $O(m) \times Lookup_{SBF}$. Además, en la estructura SBF, una vez se ha modificado el vector base, las estructuras índice necesitan ser actualizadas. Actualizar CV1 tiene un coste de $O\left(\frac{m}{\log N}\right)$, CV2 tiene un coste de $O\left(\frac{\log N}{\log\log N}\right)$, y OV un coste de $O(1)$.

Las operaciones de *rebuild* en la estructura DCF tienen un coste $O(m)$. Los *rebuild* para DCF tienen la ventaja de que iterar a través del conjunto de entradas es menos costoso debido al acceso directo que aporta la estructura, y a la explotación de la localidad de los datos en la jerarquía de memoria. Significar también que el número de *rebuilds* ejecutado es teóricamente mucho menor que el número de *rebuilds* ejecutados por SBF. Esto es debido al hecho que cada *rebuild* afecta a toda el area de *overflow* incrementando un bit a todos los contadores, mientras que en los SBF sólo se modifica un contador por operación de *rebuild*.

Los PDCF tienen el menor coste de ejecución para la operación de *rebuild*. En los PDCF, sólo aquellas particiones que tengan contadores con *overflow* requieren la operación de de *rebuild*, mostrando un coste asintótico de $O(m') = O\left(\left\lceil \frac{m}{\gamma} \right\rceil\right)$. Además, la estructura PDCF hereda el rápido aceso de los DCF. Esta mejora en el coste de la operación de *rebuild* será significativa en el rendimiento global, tanto en tiempo de ejecución como en el uso de la memoria, especialmente cuando el vector base es grande (m crece), y las operaciones de *rebuild* se encarecen en coste tanto para los DCF como para los SBF. También es importante decir que, para distribuciones no uniformes, los PDCF tienen que disponer de menos espacio de memoria que los DCF, lo cual también se ve reflejado en una reducción del tiempo total de la operación de *rebuild*. También podemos observar este hecho en la Figura 7.9-(a) donde el tiempo total de ejecución para distribuciones no uniformes se reduce de forma más drástica que para distribuciones uniformes.

7.7 Configuración de la evaluación

Para nuestra evaluación hemos implementado las técnicas SBF, DCF, y PDCF en lenguaje C. La implementación de los SBF ha seguido, de forma detallada, las especificaciones dadas en [27]. Nuestras pruebas se ejecutan en un procesador IBM Power_RS64-IV a 750 MHz con 16GB de memoria principal bajo el sistema operativo AIX 5.1.

Entorno de ejecución

Utilizamos datos generados a través de una distribución Zipfian [18], donde el sesgo en los datos viene definidio por la variable θ y puede tomar valores entre 0.0 y 2.0. Valores para $\theta \simeq 0$ representan valores uniformemente distribuidos, mientras valores para $\theta \simeq 2$ representan valores con mucho sesgo. Utilizamos números enteros para los valores de los datos.

Nuestras pruebas se centran principalmente en un entorno dinámico en el que se simula una situación donde un total de tres millones de operaciones (1 millón de inserciones, 1 millón de borrados, 1 millón de consultas) se agrupan de forma aleatoria en diferentes pasos de ejecución a lo largo del tiempo de vida de la ejecución. Para las operaciones de inserción, θ se escoge de forma aleatoria entre el rango $\{0..2\}$ en cada paso de ejecución. También, el *ratio* entre inserciones y borrados cambia a lo largo del tiempo. Los elementos borrados y consultados, también se escogen aleatoriamente en cada paso de ejecución. Una consulta consiste en acceder d contadores de cada estructura. Donde d es el número de funciones de hash.

El número de particiones γ para los PDCF se calcula usando la ecuación 7.1 explicada en la Sección 7.5.3. Los valores para el tiempo de ejecución de una operación de *rebuild* t_R y la frecuencia $freq$ se obtienen a partir de las ejecuciones reales. El número de contadores m para las tres técnicas evaluadas depende de la fracción de falsos positivos P, el número de valores distintos n, y el número de funciones de hash d utilizadas. Establecemos por defecto $P = 0.05$ y $d = 3$. El número de valores distintos n puede cambiar según la prueba, y se especifica para cada experimento que describamos.

Una de las métricas utilizadas para medir la precisión del filtro es la *calidad de representación*. Decimos que un elemento s tiene una óptima calidad de representación si el mínimo de los valores almacenados en los d contadores es igual al número real de veces que s se ha insertado en el conjunto de datos representado.

7.8 Resultados

Antes de presentar el análisis dinámico mostraremos resultados respectivos a los tiempos de acceso y el tiempo de *rebuild* para un escenario estático, donde los elementos primero se insertan, luego se consultan, y finalmente se borran. Para el escenario estático también veremos los efectos que puede tener el sesgo en los datos para SBF y DCF.

Si no se específica lo contrario para algún ejemplo en particular, todos los resultados que se muestran en las siguientes Secciones se han obtenido a partir de las ejecuciones reales efectuadas.

Lectura, escritura, y tiempo de rebuild

La Figura 7.11, muestra el tiempo medio para ejecutar una lectura, una escritura, y una operación de *rebuild* por las diferentes estructuras SBF, DCF, y PDCF, según distintos tamaños del conjunto de elementos representado. Los resultados se muestran para 4 situaciones distintas donde el número de elementos distintos n es 1000, 10000, 100000, y 1000000, respectivamente. Cada prueba consiste en una ejecución estática donde el número total de valores $M = 100 \times n$, uniformemente distribuidos, primero son insertados, luego consultados, y finalmente borrados de forma aleatoria. Los resultados se expresan en μsegundos y el eje de ordenadas se representa en escala logarítmica. Cada valor en la gráfica representa la media, para cada operación, a lo largo de todas las instancias realizadas para cada operación durante la ejecución de cada prueba.

Los tiempos de acceso de escritura y lectura para los DCF, son los mismos que para los PDCF, así que no se incluyen en la gráfica para una mayor claridad. Como se puede observar los DCF y PDCF presentan unos métodos de acceso mucho más rápidos que los SBF, reduciendo a más de la mitad el tiempo de lectura y escritura. Esto es debido al acceso directo de la estructura DCF en comparación al uso de estructuras índice de los SBF para acceder a los contadores.

Como era de esperar, PDCF muestra un clara mejora en cuanto a tiempo de *rebuild* con respecto SBF y DCF. Además, mientras que el tiempo de *rebuild* para DCF crece a la par que lo hace

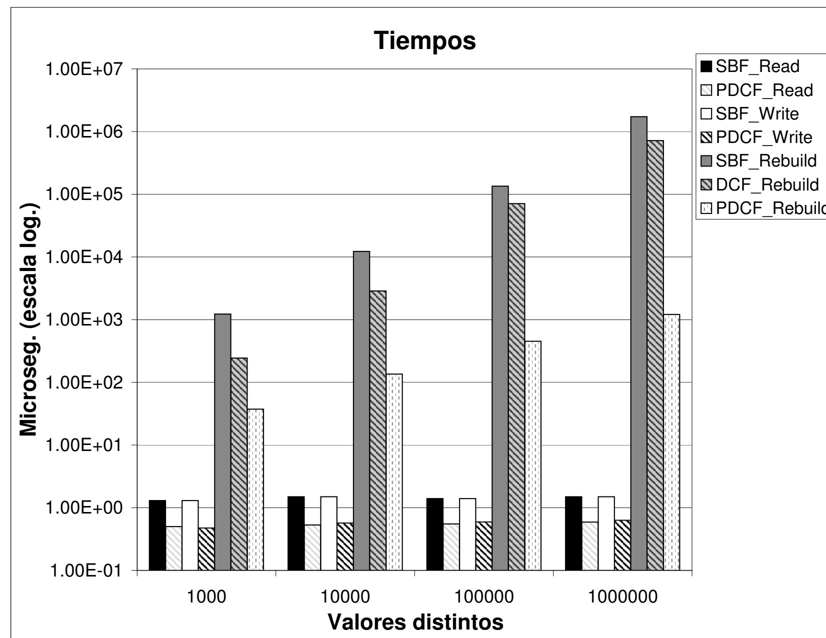


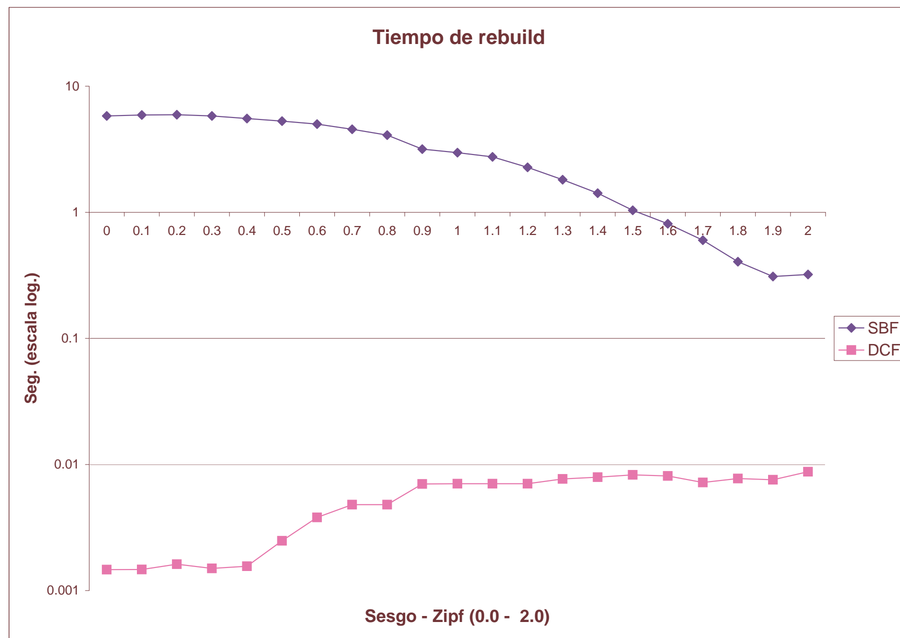
Figura 7.11 Tiempo de ejecución medio para una operación de lectura, escritura, y de *rebuild*.

SBF, para PDCF el tiempo crece de forma menos progresiva y muestra un aumento más pequeño en proporción al número de valores distintos. Un aumento en el número de valores distintos n significa que el tamaño del filtro se incrementa (el valor de m incrementa). Entonces, mientras que el coste de la operación de *rebuild* para SBF y DCF depende estrictamente de m (ver Sección 7.6), el coste de la operación de *rebuild* para PDCF también depende del número de particiones γ. Cuanto mayor es el número de valores distintos, mayor puede ser γ (véase ecuación 7.1 en Sección 7.5.3), y por lo tanto, mientras el tiempo de *rebuild* crece de forma proporcional a m para DCF y SBF, lo hace proporcional a $\frac{m}{\gamma}$ para PDCF.

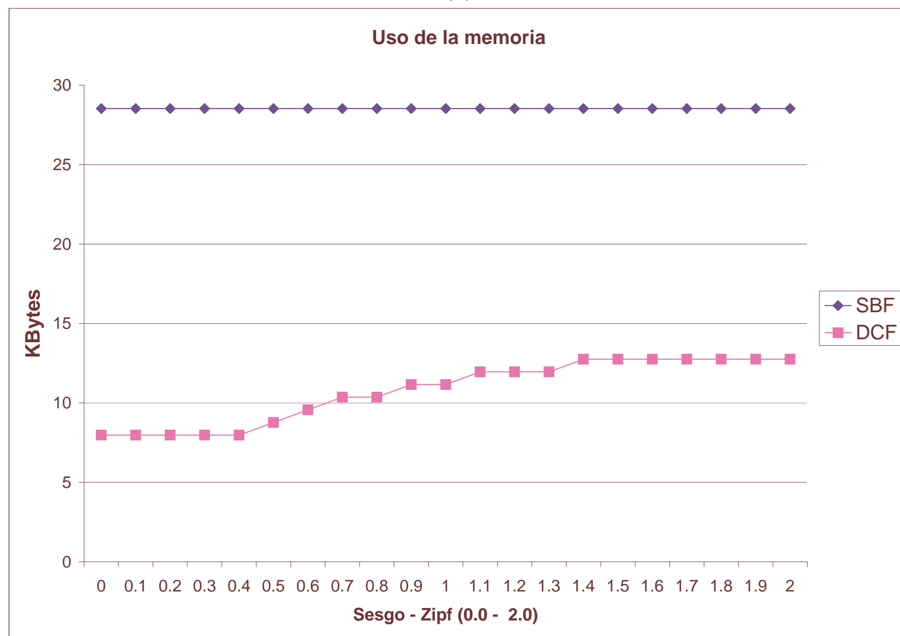
Efecto del sesgo en los datos

Para el mismo escenario estático descrito anteriormente, y cogiendo un total de 1000 valores distintos como conjunto de pruebas, las Figuras 7.12-(a) y 7.12-(b) muestran la influencia que tiene el sesgo en los datos para SBF y DCF, sobre el tiempo de *rebuild* (en escala logarítmica) y el uso de la memoria respectivamente. Los resultados para DCF son generalizables a PDCF, pues PDCF utiliza DCF para governar cada una de sus particiones.

Como se puede observar, para DCF, un mayor sesgo en los datos se traduce en un mayor consumo de memoria y mayor tiempo de *rebuild*. Esto era de esperar, pues como se explica en la Sección 7.6.1, cuando hay mucho sesgo en los datos, la estructura puede alcanzar su tamaño máximo totalizando $\log(M)$ bits por contador. A su vez, el tiempo de *rebuild* aumenta debido al mayor tamaño de la zona comprendida por el vector de *overflow*. Por otra parte, el tiempo de *rebuild* mejora para SBF cuando hay sesgo debido a que se realiza un menor número de operaciones de *rebuild*, pudiendo así sacar provecho de los *slack bits* más cercanos. El uso de la memoria para SBF es siempre constante para SBF independientemente del sesgo en los datos (véase Sección 7.6.1).



(a)

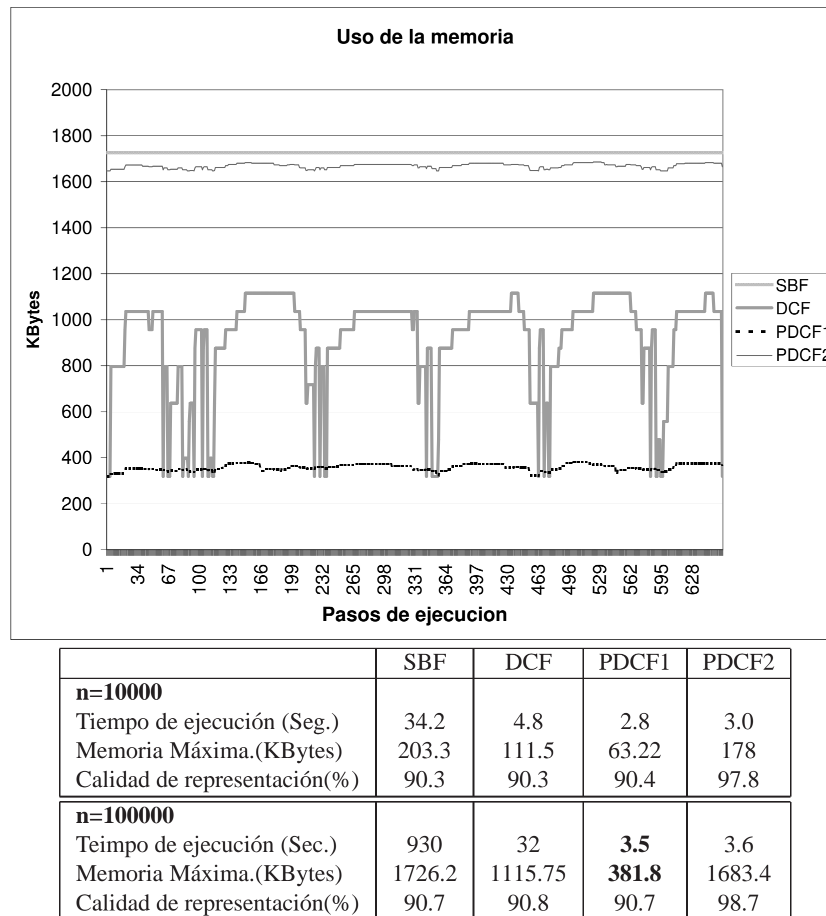


(b)

Figura 7.12 (a) Tiempo de ejecución; (b) Uso de la memoria.

Análisis dinámico

El análisis dinámico muestra el comportamiento de SBF, DCF, y PDCF, en un escenario en el que los valores se insertan, borran y consultan de forma aleatoria a lo largo del tiempo. Como se



	SBF	DCF	PDCF1	PDCF2
n=10000				
Tiempo de ejecución (Seg.)	34.2	4.8	2.8	3.0
Memoria Máxima.(KBytes)	203.3	111.5	63.22	178
Calidad de representación(%)	90.3	90.3	90.4	97.8
n=100000				
Teimpo de ejecución (Sec.)	930	32	**3.5**	3.6
Memoria Máxima.(KBytes)	1726.2	1115.75	**381.8**	1683.4
Calidad de representación(%)	90.7	90.8	90.7	98.7

Figura 7.13 Resultados del análisis dinámico.

ha mencionado previamente, el análisis dinámico se compone de tres millones de operaciones: 1 millón de inserciones, 1 millón de borrados, y un millón de consultas. Se han realizado las pruebas con un número de valores distintos $n = 10000$ y $n = 100000$.

Con el fin de obtener una comparación justa entre las tres representaciones, consideramos dos configuraciones distintas para PDCF:

- **PDCF1**. En este caso no hay cambios con la configuración base explicada para PDCF. PDCF1, SBF y DCF tienen la misma probabilidad de falsos positivos P y utilizan el mismo número de funciones de hash d. Consecuentemente, las tres representaciones pueden representar el conjunto de datos con la misma precisión.

- **PDCF2** (*Cantidad de memoria limitada*). En esta configuración la ejecución de PDCF es forzada a utilizar la misma cantidad de memoria consumida por SBF. El espacio extra se utiliza para decrementar la probabilidad de falsos positivos P, y en consecuencia, el número de contadores para PDCF se ve incrementado. El resultado esperado de esta configuración reside en que, utilizando la misma cantidad de memoria que la utilizada por SBF, PDCF representará el conjunto de datos con una mayor precisión que SBF y DCF.

Además de la gráfica de la Figura 7.13 que representa el uso de la memoria para las técnicas evaluadas en el caso de 100000 valores distintos, también proveemeos un resumen de los resultados en la Tabla situada debajo de la misma Figura.

La primera conlcusión que podemos extraer de los resultados experimentales, es que para el mismo número de operaciones, DCF y PDCF completan su ejecución mucho más rápidamente que SBF. A la vez, PDCF resulta más rapido que DCF. En particular, para $n = 100000$, mientras PDCF completa su ejecución en 3.5 segundos, SBF requiere 930 segundos y DCF 32 segundos. La mejora de DCF y PDCF respecto a los SBF se justifica a partir de la gráfica mostrado en la figura 7.11, en el que se demuestra la clara superioridad en cuanto a tiempos de acceso y tiempo de *rebuild* de los DCF y PDCF respecto a los SBF. La diferencia entre PDCF y DCF se justifica a partir de la clara mejora en el tiempo de *rebuild* por parte de los PDCF, también observable en la Figura 7.11.

La gráfica en la Figura 7.13 muestra el consumo de memoria por parte de las tres estrategias a lo largo del tiempo de ejecución para $n = 100000$. Aunque Mem_{PDCF} y Mem_{DCF} fluctuan debido a la variabilidad dinámica introducida por el límite T autoconfigurable (véase Sección 7.4), podemos observar que la memoria consumida por PDCF1 es casi tres veces menor que la memoria consumida por DCF y casi 6 veces menor que la consumida por SBF. Los elementos se insertan y borran a lo largo del tiempo, de modo que, mientras SBF tiene un tamaño estático ya desde el principio, DCF y PDCF adaptan dinámicamente su tamaño a los requerimientos del conjunto de datos representado. Consecuentemente, tanto PDCF1 como DCF ahorran en memoria obteniendo la misma calidad de representación que SBF.

Los resultados obtenidos para PDCF2 muestran que PDCF, consumiendo la misma memoria que SBF, obtiene una calidad de representación del 98.4%, lo cual significa un 8% de mejora respecto los DCF y SBF. También se han realizado ejecuciones de los DCF con la misma cantidad de memoria que la utilizada por SBF. Para estas pruebas, la calidad de representación de los DCF es de un 95%, lo cual es un 3% menor que PDCF2.

Calidad de servicio

La Figura 7.14 muestra, para la ejecución con $n = 100000$ analizada anteriormente, como la mejora en el tiempo de ejecución se ve reflejada en la calidad de servicio. El eje de ordenadas (en escala logarítimica) indica el tiempo acumulado de respuesta para las consultas realizadas a lo largo del tiempo de ejecución. PDCF es capaz de responder un millón de consultas en menos de un segundo en contraste con los 5.5 segundos de DCF y los 142 de SBF.

n	$t_R(\mu seg)$	$S_e(bytes)$	$S_p(bytes)$	$S_c(bytes)$
1000	244	4	4	2

$freq.(\frac{peticiones}{segundo})$	10^5	10^6	10^7	10^8
γ	9	28	90	285
Buffer area (KB)	0.1175	0.0703	0.01558	0.0127
$(\frac{throughput_{1-particion}}{throughput_{\gamma-particiones}})$	2.02	33.8	1529.4	18762.3

Tabla 7.2 Resultados para PDCF.

La Tabla 7.2, de acuerdo con el modelo presentado en la Sección 7.5.3, muestra como la mejora en el tiempo de *rebuild* afecta tanto al tamaño de la *buffer area* como al *throughput* comparado con DCF ($throughput_{1-particion}$). Mostramos estimaciones para diferentes frecuencias de los *streams*

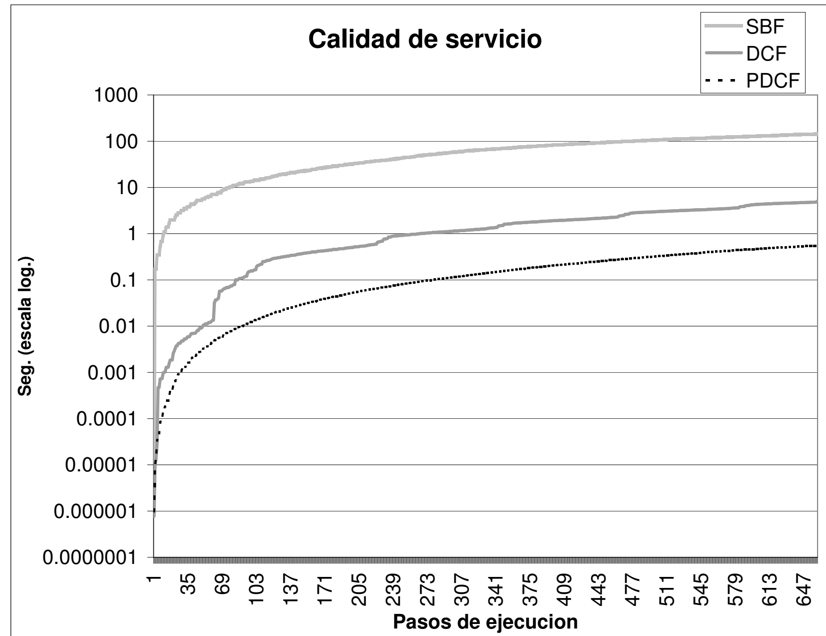
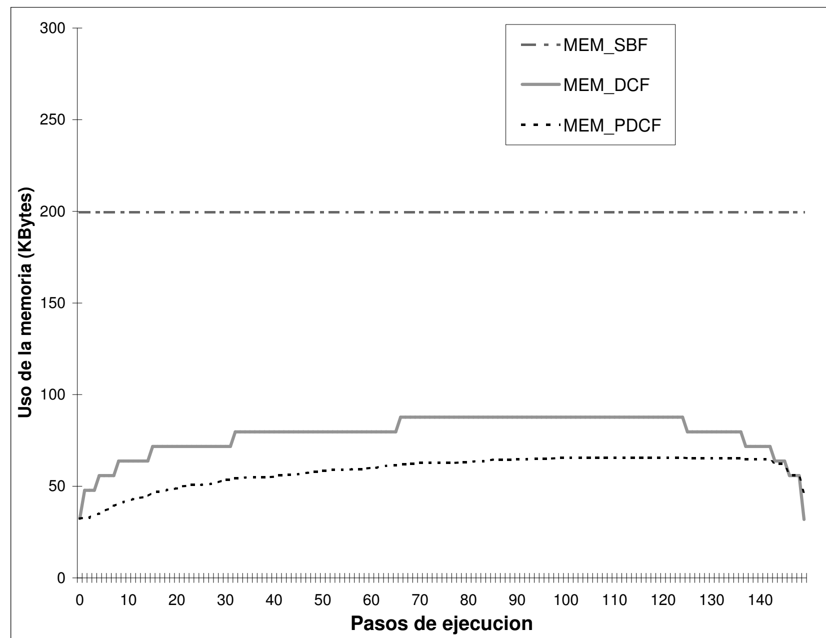


Figura 7.14 Calidad de servicio. Tiempo requerido para responder consultas a lo largo del tiempo de ejecución para cada una de las 3 estrategias evaluadas.

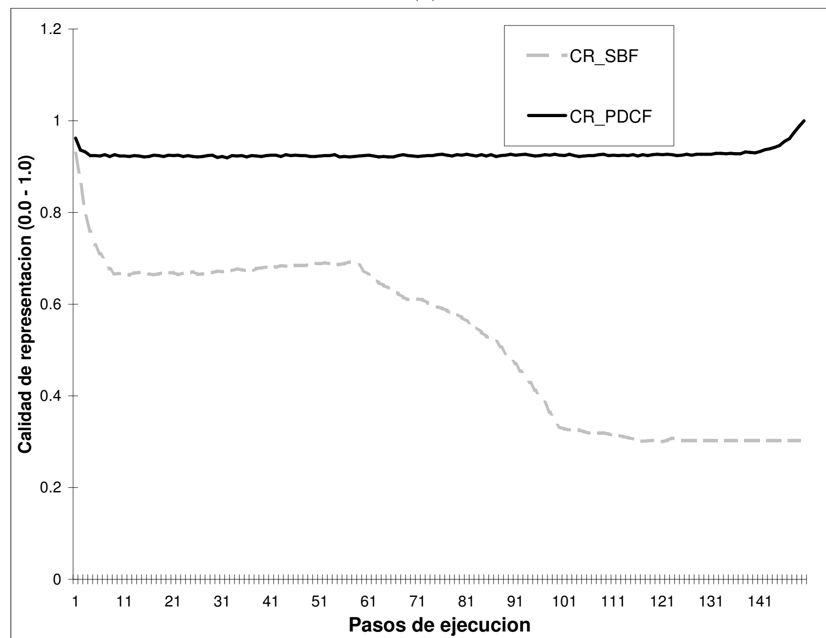
de datos entrantes $freq$, desde 10^5 hasta 10^8 peticiones por segundo. El número de particiones γ para cada frecuencia se determina utilizando la ecuación 7.1, los elementos se consideran de 4 bytes, $S_e = 4$. También, establecemos el tamaño de un contador en 2 bytes $S_c = 2$ y el tamaño de un puntero en 4 bytes $S_p = 4$. t_R es el tiempo de *rebuild* observado para una partición $\gamma = 1$, equivalente a la estrategia DCF. Los resultados relativos al *throughput* muestran que cuanto mayor es la frecuencia, mayor es la mejora debido a la posibilidad de realizar más particiones, y, en consecuencia, el tiempo de *rebuild* t_R se reduce de forma proporcional. El tamaño de la *buffer area* también se reduce debido al bajo tiempo de *rebuild*. Sin embargo, la mejora obtenida en el *throughput* no es tan significativa porque el aumento de la frecuencia contraresta la mejora causada por el tiempo de *rebuild*.

Baja estimación del conjunto de datos representado

En esta Sección presentamos resultados donde forzamos las tres estructuras para un caso en el que tenemos muchas más operaciones de inserción que las que se esperaban cuando las estructuras de datos fueron creadas. Así pues, creamos los SBF, DCF, y PDCF para representar un conjunto de 100000 elementos, posteriormente insertamos un total de 500000 elementos en las 3 estructuras, y finalmente borramos aleatoriamente todos los datos. Los elementos insertados se separan previamente en diferentes grupos que son, posteriormente insertados en cada paso del tiempo. Los elementos insertados para cada grupo siguen una distribución Zipfian, donde el sesgo θ varía de forma aleatoria entre grupos. El número de valores distintos es $n = 10000$. Este escenario representa un caso especial de un inesperado número de elementos entrantes en algún momento en el tiempo de vida de una aplicación.



(a)



(b)

Figura 7.15 (a)Uso de la memoria; (b) calidad de representación (CR_PDCF = CR_DCF).

La Figura 7.15-(a) muestra el uso de la memoria, y la Figura 7.15-(b) muestra la calidad de representación para las tres estrategias. Ambas figuras muestran resultados para los diferentes pasos en el tiempo dentro del escenario puesto a prueba y explicado con anterioridad. Para mayor

claridad, la calidad de representación de los DCF no se muestra al ser la misma que la obtenida por los PDCF. A través de estos resultados podemos observar que los PDCF heredan la flexibilidad de los DCF comparado con los SBF. En el escenario probado, los SBF tienen un uso de la memoria constante, el predefinido en su creación. De este modo, a medida que se insertan los elementos, su calidad de representación se mantiene alta hasta el punto en el que el número de elementos insertados sobrepasa el esperado. Llegado este punto, los contadores en la estructura SBF se saturan y en consecuencia su calidad de representación disminuye drásticamente. Por el contrario, la calidad de representación de los PDCF es prácticamente constante y casi siempre se mantiene alta. Esto se consigue debido a la habilidad que los DCF y PDCF tienen de incrementar dinámicamente el area de *overflow*. Este hecho se muestra en el incremento del uso de memoria realizado por DCF y PDCF.

En definitiva, podemos concluir que, mientras SBF tiene que ser muy precisa al dimensionar su tamaño, siendo incapaz de adapatarse a situaciones inesperadas, los DCF y PDCF son flexibles y capaces de manejar cualquier tipo de situación extrema del conjunto de datos que se representa. Además, PDCF mantiene un *throughput* alto independientemente del tamaño del conjunto de datos representado, y es más eficiente en espacio que los DCF.

7.9 Conclusiones

En este Capítulo hemos presentado los *Dynamic Count Filters* (DCF) y su versión particionada, los *Partitioned Dynamic Count Filters* (PDCF). Ambos son representaciones dinámicas de los *Count Bloom Filter* (CBF), con rápidos métodos de acceso y operaciones de *rebuild* de bajo coste. Todo ello se traduce en una mejora del tiempo total de ejecución y en una mejor calidad de servicio respecto a los *Spectral Bloom Filters* (SBF), que son la aproximación dinámica de los CBF propuesta en la literatura, y más cercana a nuestro trabajo.

Los PDCF son una optimización de los DCF, que reducen aún más, el uso de la memoria y el coste de la operación de *rebuild*, mejorando así el rendimiento global de la estructura. En concreto (i) la cantidad de memoria se reduce a la mitad comparado con los DCF, y en más de una tercera parte comparado con los SBF, y (ii) el tiempo de ejecución se reduce en un orden de magnitud comparado con los DCF y dos ordenes de magnitud comparado con los SBF.

Además, los PDCF se adaptan de forma simple a las distribuciones no uniformes y a los cambios dinámicos de los datos, aportando pequeñas variaciones en la memoria utilizada, y rápidos accesos a la estructura, de una forma significativamente más eficiente que los SBF y DCF.

8

CONCLUSIONES

En esta Tesis hemos abordado técnicas para la mejora de la operación de join paralela, y el procesamiento de secuencias temporales de datos. En la Figura 8.1 se muestra qué técnicas cumplen con los objetivos que nos habíamos propuesto en nuestro trabajo de investigación.

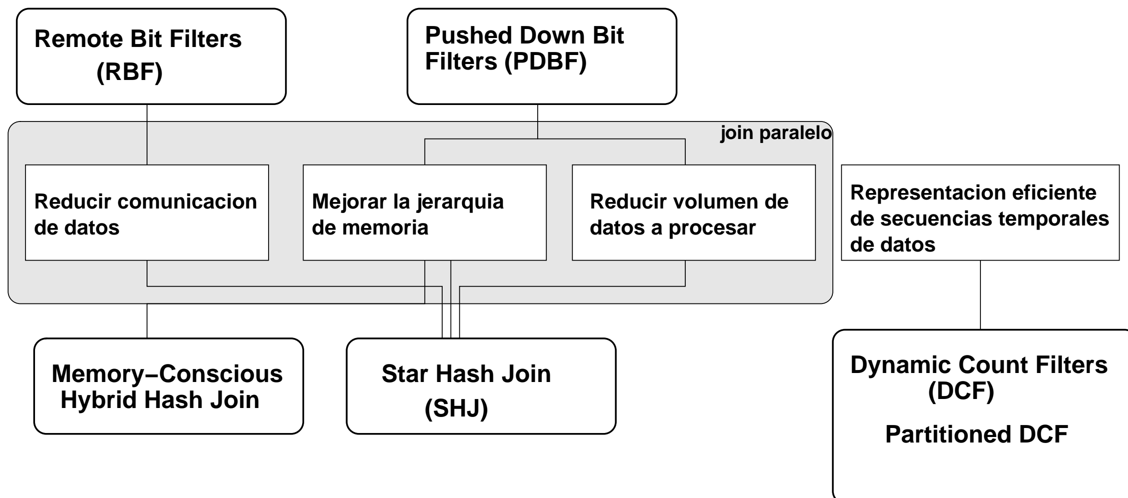


Figura 8.1 Objetivos y técnicas.

Tanto en el area de las bases de datos como el de las secuencias temporales de datos, hemos explotado el uso de los *bit filters* con la siguiente finalidad:

- reducir la comunicación de datos en sistemas distribuidos durante la operación de join. En concreto hemos propuesto la técnica *Remote Bit Filters with Requests* (RBF_R), que, a través de un sistema de peticiones, realiza un uso remoto de los *bit filters* siendo eficiente en los recursos de memoria utilizados. Hemos demostrado que en un amplio abanico de escenarios, RBF_R obtiene un rendimiento similar a los *Remote Bit Filter Broadcasted* (RBF_B), evitando los problemas que esta última presenta durante la ejecución de múltiples operaciones de join en paralelo. Los resultados para nuestras pruebas muestran un beneficio de hasta un 27% en

tiempo de ejecución respecto la ejecución original de el Sistema Gestor de Bases de Datos de IBM, DB2 UDB, reduciendo en más de un 40% la carga de la capa de comunicación.

- reducir el volumen de datos a procesar por el plan de ejecución de una consulta. Para ello hemos propuesto la técnica *Pushed Down Bit Filters* (PDBF), que, utilizando los bit filters generados en los nodos superiores del plan de ejecución, elimina los datos desde los nodos inferiores del plan a sabiendas que estos datos serán eliminados en operaciones futuras. Además la implementación de la técnica requiere de pocos recursos adicionales mostrando un buen rendimiento bajo cualquiera de las configuraciones probadas, y obteniendo una mejora de hasta más de un 50% en tiempo de ejecución para los mejores casos.

- obtener una representación eficiente de secuencias temporales de datos. Este objetivo se cumple con la propuesta de los *Dynamic Count Filters* (DCF), y de su versión particionada, los *Partitioned DCF* (PDCF). Ambas técnicas se basan en unas estructuras de datos dinámicas, adaptables, y con rápidos métodos de acceso, que dan una representación de un conjunto de datos que evoluciona a través del tiempo. Los DCF y PDCF, dentro de nuestro conocimiento, mejoran en términos de rendimiento y consumo de memoria respecto a cualquier otra propuesta previa en la literatura.

Los RBF y PDBF se combinan en la técnica *Star Hash Join* (SHJ) para explotar el uso de los *bit filters* con el fin de acelerar la ejecución paralela de la operación de join estrella en sistemas *cluster*. Con esta técnica se consigue disminuir la comunicación de datos entre nodos casi al mínimo posible en un amplio repertorio de casos. Además disminuye la E/S, y el volumen de datos a procesar por el plan de ejecución gracias al efecto de la técnica PDBF, que se extiende en este caso a sistemas *cluster*.

También, con el fin de mejorar el paralelismo *intra-cluster*, hemos presentado la técnica *Memory-Conscious Hybrid Hash Join*, que consiste en un algoritmo paralelo para la fase de join de dicha operación. El algoritmo se adapta en tiempo de ejecución a los recursos disponibles por la operación de join, disminuyendo la contención de memoria, sincronización entre procesos, y operaciones de E/S.

En definitiva, podemos concluir que las técnicas presentadas abarcan los objetivos de esta Tesis, mejorando la operación de join paralelo a través de (i) el ahorro de comunicación de datos, (ii) una reducción del volumen de datos a procesar, y (iii) un mejor uso de la memoria, y su consecuente ahorro en operaciones de E/S. Finalmente hemos extendido el uso de los *bit filters* al area de las secuencias temporales de datos, y hemos aportado técnicas que nos dan una representación eficiente en espacio y tiempo de conjuntos de datos que evolucionan a través del tiempo.

Trabajo futuro

Las lineas de investigación futuras estan centradas en los siguientes puntos:

- La validación del modelo analítico presentado para la técnica *Star Hash Join*. Pese a estar basado en modelos ya validados como el de los *Remote Bit Filters*, queremos introducir un prototipo de esta técnica en un Sistema Gestor de Bases de Datos, y contrastar los resultados reales con los obtenidos por el modelo presentado en esta Tesis.

- Desarrollo de nuevos algoritmos paralelos para el *Hybrid Hash Join*. Creemos que hay mucha investigación a realizar dentro de la ejecución de dicho algoritmo en sistemas paralelos con multiprocesadores simétricos. Basándonos en el trabajo desarrollado, querríamos

contemplar la posibilidad de incluir nuevos algoritmos que pudieran obterner un mejor rendimiento en situaciones en las que los datos están sesgados por múltiples valores, y por lo tanto, podemos tener varios buckets con tamaños dispares.

- Validación del modelo matemático para el *Memory-Conscious Hybrid Hash Join* cuando hay sesgo en los datos.

- Paralelismo y aplicaciones de los *Dynamic Count Filters*. Queremos investigar el rendimiento de la estructura de datos bajo entornos paralelos, así como su aplicación en configuraciones reales, ya sean hardware o software.

- Probar la versión particionada de los *Spectral Bloom Filters*, y compararla con la técnica *Partitioned Dynamic Count Filters*. También evaluar y comparar nuestras técnicas con nuevos algoritmos aparecidos en recientes estudios [68] y que se presentan como un reemplazo de los *Bloom Filters*.

REFERENCIAS

[1] www.Top500.org.

[2] www.tpc.org.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370, 2004.

[4] J. Aguilar-Saborit, V. Muntés-Mulero, and J.-L. Larriba-Pey. Pushing Down Bit Filters in the Pipelined Execution of Large Queries. *Euro-Par'03: Parallel Processing, 9th International Euro-Par Conference*, pages 328–337, 2003.

[5] J. Aguilar-Saborit, V. Muntes-Mulero, J.-L. Larriba-Pey, C. Zuzarte, and H. Pereyra. A data traffic memory efficient protocol for the use of bit filter in shared nothing partitioned systems. *XV Jornadas de Paralelismo*, pages 444–449, 2003.

[6] J. Aguilar-Saborit, V. Muntes-Mulero, J.-L. Larriba-Pey, C. Zuzarte, and H. Pereyra. On the use of bit filters in shared nothing partitioned systems. *IWIA'05: International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems*, pages 29–37, 2005.

[7] J. Aguilar-Saborit, V. Muntes-Mulero, P. Trancoso, and J.-L. Larriba-Pey. Dynamic Count Filters. *SIGMOD Rec.*, 35(1):26–32, 2006.

[8] J. Aguilar-Saborit, V. Muntes-Mulero, A. Zubiri, C. Zuzarte, and J.-L. Larriba-Pey. Dynamic out of core join processing in symmetric multiprocessors. *PDP'06: 14th Euromicro International Conference on Parallel, Distributed and Network based Processing, France, 2006*.

[9] J. Aguilar-Saborit, V. Muntes-Mulero, C. Zuzarte, and J.-L. Larriba-Pey. Ad-hoc star hash join processing in Clusters Architectures. *DAWAK'05: 7th International Conference in Data Warehousing and Knowledge Discovery*, pages 200–209, 2005.

[10] C. Baru, G. Fecteau, A. Goyal, H.-I. Hsiao, A. Jhingran, S. Padmanabhan, and W. Wilson. DB2 Parallel Edition. *IBM Systems journal*, 1995.

[11] R. Bayer. The Universal B-Tree for Multidimensional Indexing: general Concepts. *WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications*, pages 198–209, 1997.

[12] P. A. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM*, 28(1):25–40, 1981.

[13] P. A. Bernstein and N. Goodman. The power of natural joins. *SIAM J.Computi.*, 10:751–771, November 1981.

[14] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] P. M. Bober and M. J. Carey. On Mixing Queries and Transactions via Multiversion Locking. *ICDE '92: Proceedings of the 8th International Conference on Data Engineering*, pages 535–545, 1992.

[16] P. M. Bober and M. J. Carey. Managing Memory to Meet Multiclass Workload Response Time Goals. *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 328–341, 1993.

[17] K. Bratbergsengen. Hashing methods and relational algebra operations. *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, 1984.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *In Proceedings of the IEEE Infocom Conference*, 1999.

[19] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A survey. *A survey. In Proc. of Allerton Conference*, 2002.

[20] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 397–410, 1989.

[21] M. J. Carey and W. Muhanna. The Performance of Multiversion Concurrency control Algorithms. *ACM Trans. on Computer Systems*, 4(4):338–378, 1986.

[22] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 355–366, 1998.

[23] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps. The Starfire SMP interconnect. *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, 1997.

[24] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[25] S. Chaudhuri and U. Dayal. Data warehousing and OLAP for decision support. *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 507–508, 1997.

[26] M.-S. Chen, H.-I. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *The VLDB Journal*, 6(2):121–131, 1997.

[27] S. Cohen and Y. Matias. Spectral bloom filters. *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252, 2003.

[28] G. Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD Rec.*, 25(3):64–69, 1996.

[29] D. E. Culler. Clusters of Symmetric Multiprocessors. *Final Report for Microproject 97-037. Computer Science Division. University of California (Berkeley). Industrial Sponsors: Sun Mycrosystems Corp, Adaptec Inc.*, 1998.

[30] D. L. Davison and G. Graefe. Memory-Contention Responsive Hash Joins. *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 379–390, 1994.

[31] P. Deshpande, K. Ramasamy, A. Shuckla, and J. F. Naughton. Caching multidimensional queries using chunks. *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 259–270, 1998.

[32] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases*, pages 151–164, August, 1985.

[33] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, 1986.

[34] D. J. DeWitt, S. Ghanderaizadeh, and D. Schneider. A performance analysis of the gamma database machine. *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 350–360, 1988.

[35] D. J. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[36] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 1–8, 1984.

[37] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212, 2003.

[38] X. Du, X. Zhang, Y. Dong, and L. Zhang. Architectural effects of symmetric multiprocessors on tpc-c commercial workload. *J. Parallel Distrib. Comput.*, 61(5):609–640, 2001.

[39] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 75–80, 2001.

[40] L. Fan, , P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE Trans on Networking*, 8(3):281–293, 2000.

[41] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 299–310, 1998.

[42] P. Gongloor and S. Patkar. Hash joins: Implementation and tuning, release 7.3. Technical report, Oracle Technical Report, March 1997.

[43] J. Goodman. An Investigation of Multiprocessors Structures and Algorithms for Database Management. Technical report ucb/erl, m81/33, University of California at Berkeley, May 1981.

[44] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[45] G. Graefe. Sort-Merge-Join: An Idea Whose Time Has(h) Passed? *ICDE '94: Proceedings of the 10th International Conference on Data Engineering*, pages 406–417, 1994.

[46] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 86–97, 1998.

[47] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 168–177, 1991.

[48] N. Karayannidis, A. Tsois, T. K. Sellis, R. Pieringer, V. Markl, F. Ramsak, R. Fenk, K. Elhardt, and R. Bayer. Processing star queries on hierarchically-clustered fact tables. *VLDB'02: 28th International Conference on Very Large Data Bases*, pages 730–741, 2002.

[49] A. Kemper, D. Kossmann, and C. Wiesner. Generalised hash teams for join and group-by. *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 30–41, 1999.

[50] Kitsuregawa, M., Tanaka, H., and T. Moto-oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing, Vol 1, No 1*, 1983.

[51] J. Ledlie, L. Serban, and D. Toncheva. Scaling filename queries in a large-scale distributed file systems. (TR-03-02), January 2002.

[52] Z. Li and K. A. Ross. Better Semijoins using Tuple Bit-vectors. *Columbia University Technical Report CUCS-010-94*, April 1994.

[53] S. S. Lightstone, G. Lohman, and D. Zilio. Toward autonomic computing with DB2 universal database. *SIGMOD Rec.*, 31(3):55–61, 2002.

[54] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers. *VLDB'90: 16th International Conference on Very Large Data Bases*, August, 1990.

[55] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. *VLDB'86 Twelfth International Conference on Very Large Data Bases*, pages 149–159, August, 1986.

[56] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. *VLDB'02: In Proceedings of the 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

[57] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. *Proc. of the Intl. Database Enfineering and Applications Symposium*, pages 165–177, 1999.

[58] M. Mehta and D. J. DeWitt. Dynamic Memory Allocation for Multiple-Query Workloads. *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 354–367, 1993.

[59] M. Mehta and D. J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 382–394, 1995.

[60] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking.*, 10(5):604–612, 2002.

[61] J. K. Mullin. A second look at bloom filters. *Commun. ACM*, 26(8):570–571, 1983.

[62] Nikos and T. Sellis. Sisyphus: A chunk-based storage manager for olap cubes. *DMSW '2001: Proceeding of the 23rd International Workshop on Design and Management of Data Warehouses*, 2001.

[63] E. Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 375–385, 1991.

[64] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, 1995.

[65] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. *ACM SIGMOD International conference on Mangement of Data.*, pages 38–49, 1997.

[66] S. Padmanabhan, T. Malkemus, B. Bhattacharjee, M. Huras, L. Cranston, and T. Lai. Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2. *VLDB'03: Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[67] S. Padmanabhan, T. Malkemus, B. Bhattacharjee, M. Huras, L. Cranston, and T. Lai. Multi-Dimensional Clustering: a new data layout scheme in DB2. *SIGMOD '03: ACM SIGMOD International conference on Mangement of Data.*, 2003.

[68] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829, 2005.

[69] H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. pages 59–68, 1993.

[70] PostgreSQL. http://www.postgresql.org/.

[71] X. Qin and J.-L. Baer. A performance evaluation of cluster architectures. *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 237–247, 1997.

[72] E. Rahm and R. Marek. Dynamic Multi-Resource Load Balancing in Parallel Database Systems. *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 395–406, 1995.

[73] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, 1989.

[74] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569, 2002.

[75] D. a. R. E. Ries. Evaluation of Distribution Criteria for Distributed Database Systems. *UCB/ERL Technical Report M78/22, UC Berkeley, May, 1978.*

[76] R. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Rec.*, 27(1):21–26, 1998.

[77] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 110–121, 1989.

[78] S. Sethimadhavan, R. Desikan, and D. Burger. Scalable hardware memory disambiguation for high ilp processors. *MICRO '03: Proc. of the 36th International Symposium on Michroarchitecture*, 2003.

[79] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[80] C. Walton, A. G. Dale, and R. M. Jenevin. A taxonomy and performance mode of data skew effects in parallel joins. *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 537–548, September, 1991.

[81] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 240–251, 2004.

[82] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[83] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit Transposed Files. *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases*, pages 448–457, August 21-23 1985.

[84] S. B. Yao. Approximating Block Accesses in Database Organizations. *Communications of the ACM*, 20, no.4:260–261, April 1977.

[85] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. *VLDB'90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 186–197, 1990.

[86] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1087–1097, 2004.

A

MODELO ANALÍTICO PARA LOS REMOTE BIT FILTERS

Proponemos un modelo analítico para *Remote Bit Filters with Requests* (RBF$_R$), *Remote Bit Filters Broadcasted* (RBF$_B$), y el uso local de los *bit filters* (*baseline*). Estos modelos son aplicables a cualquier SGBD sobre una arquitectura *cluster* como la descrita a lo largo de esta Tesis.

Basamos nuestros modelos bajo la suposición de que: (i) la base de datos está particionada horizontalmente entre nodos siguiendo un esquema de particionado hash, y que la relación de *probe* se reparticiona de forma selectiva durante la ejecución del algoritmo *Hybrid Hash Join* (HHJ) no colocalizado, (ii) el número de *buffers* de datos consecutivos enviados está limitado por la capacidad del operador de reparticionado que actúa como *receiving end*, (iii) los datos tienen una distribución uniforme, y en consecuencia la selectividad de los *bit filters* es la misma en cualquier nodo del sistema, y (iv) el tamaño de los *bit filters* que RBF$_B$ tiene que enviar al principio de la fase de *probe* del algoritmo HHJ, es despreciable si lo comparamos con la cantidad de datos a reparticionar.

Empezamos dando la definición de valores que el optimizador del SGBD debería ser capaz de calcular, o extraer del catálogo:

S_{bf} , selectividad de los *bit filters*.

E_{bf} , eficiencia de los *bit filters*, $1 - S_{bf}$.

N , número de tuplas por nodo involucradas en el proceso de reparticionado.

$|B_d|$, número de tuplas que caben en un *buffer* de datos.

$|B_{hc}|$, número de códigos de hash que caben en un *buffer* de datos.

$|B_{be}|$, número de entradas de un *bitmap* que caben en un *buffer* de datos.

B_{d_s} , tamaño, en bytes, de un *buffer* de datos.

B_{c_s} , tamaño, en bytes, de un *buffer* de mensajes.

t_d , tiempo medio de llegada de las tuplas al operador de reparticionado.

t_{pack} , tiempo transcurrido en empaquetar una tupla en un *buffer* .

t_{unpack} , tiempo transcurrido en desempaquetar una tupla en un *buffer* .

t_{hc} , tiempo transcurrido en construir un código de hash.

$t_{pack_{hc}}$, tiempo transcurrido en empaquetar una código de hash en un *buffer* .

$t_{unpack_{hc}}$, tiempo transcurrido en desempaquetar un código de hash en un *buffer* .

t_{test} , tiempo transcurrido en chequear un código de hash en un bit filter.

t_{be} , tiempo transcurrido en actualizar una entrada de un *bitmap*.

$t_{process_f}$, tiempo transcurrido en procesar un tupla filtrada por los nodos superiores al operador de reparticionado en el plan de ejecución. Este proceso incluye el tiempo de construir un código de hash por operador de join y el tiempo empleado en testear una tupla en el bit filter.

$t_{process_{nf}}$, tiempo transcurrido en procesar un tupla que no ha sido filtrada. Este proceso incluye el tiempo de construir un código de hash por el operador de join, y el tiempo empleado en testear una tupla en el bit filter.

t_{filter} , tiempo transcurrido en borrar un tupla de un *buffer* por el operador de reparticionado.

C , capacidad, en número de *buffers*, de un operador de reparticionado.

λ , tiempo medio en enviar un byte a través de la red de comunciación.

A continuación, para cada una de las tres estrategias (*baseline*, RBF$_R$, y RBF$_B$), definimos el coste de los tres componentes afectados: (i) el operador de reparticionado que actúa como *sending end*, (ii) el operador de reparticionado que actúa como *receiving end*, y (iii) la capa de comunicación.

Sending end t_{write}

Para la estrategia *baseline*, el operador de reaparticionado en el *sending end*, procesa una a una (t_d) las tuplas suministradas por los nodos inferiores del plan, y las empaqueta en el *buffer* de datos (t_{pack}) asociado al nodo destino. El coste asociado a esta operación es:

$$t_{write_{baseline}} = N(t_d + t_{pack}) \tag{A.1}$$

El operador de reparticionado para RBF$_R$, empaqueta cada tupla recibida ($t_d + t_{pack}$). A su vez, por cada tupla crea su respectivo código de hash y lo empaqueta en un *buffer* ($t_{hc} + t_{pack_{hc}}$), que será enviado posteriormente. Finalmente, dependiento del *bitmap* recibido por el *receiving end*, algunas tuplas serán filtradas (con el coste de borrar una tupla de un *buffer* t_{filter}), y las tuplas no filtradas son enviadas. La cantidad total de trabajo asociado a esperar las tuplas procesadas, crear y empaquetar los códigos de hash, y empaquetar las tuplas es:

$$t_{write_{RBFR}} = N(t_d + t_{pack} + t_{hc} + t_{pack_{hc}} + t_{filter}E_{bf}) \tag{A.2}$$

RBF$_B$ es similar a RBF$_R$ con la diferencia de que no se han de empaquetar los códigos de hash, y no hay que borrar tuplas de los *buffers* de datos. Sin embargo, los *bit filters* tienen que ser chequeados para cada código de hash:

$$t_{write_{RBFB}} = N\left(t_d + t_{hc} + t_{test} + t_{pack}(S_{bf})\right) \tag{A.3}$$

Receiving end t_{read}. Para la estrategia *baseline*, el *receiving end* desempaqueta las tuplas, las chequea en el bit filter local, y las procesa una a una. El coste asociado es:

$$t_{read_{baseline}} = N\left(t_{unpack} + t_{process_f}E_{bf} + t_{process_{nf}}(S_{bf})\right) \tag{A.4}$$

Para RBF$_R$, el proceso es como sigue. Primero, el *receiving end* recibe los códigos de hash, los desempaqueta, y crea el *bitmap* ($t_{unpack_{hc}} + t_{test} + t_{be}$) que se envia al *sending end*. Finalmente, recibe las tuplas no filtradas que tiene que desempaquetar y procesar ($t_{unpack} + t_{process_{nf}} - t_{test}$). Significar que $t_{process_{nf}}$ contiene el tiempo de chequear un bit filter, acción que no se debe de reejecutar en el *receiving end*. El coste asociado es:

$$t_{read_{RBFR}} = N\left(t_{unpack_{hc}} + t_{test} + t_{be} + (t_{unpack} + t_{process_{nf}} - t_{test})(S_{bf})\right) \tag{A.5}$$

RBF$_B$ vuelve a ser similar a RBF$_R$, pero los códigos de hash no tienen que ser recibidos ni procesados. Así pues, el coste para el *receiving end* es:

$$t_{read_{RBFB}} = N(t_{unpack} + t_{process_{nf}} - t_{test})(S_{bf}) \tag{A.6}$$

Capa de comunicación t_{comm}.

Para la estrategia *baseline* se envia un número de *buffers* igual al número de tuplas N divido por el número de tuplas por *buffer* $|B_d|$, a su vez, se envía un mensaje de control cada cierto número de *buffers*, determinado por una cierta capacidad C del *receiving end*. Cada byte de estos *buffers* se envía en un tiempo igual a λ. Así el coste de la capa de comunicación para la estrategia *baseline* es:

$$t_{comm_{baseline}} = N\left(\frac{1}{|B_d|}B_{d_s} + \frac{1}{|B_d|C}B_{c_s}\right)\lambda \tag{A.7}$$

En el caso de RBF$_R$ el coste de la comunciación puede ser resumido de la siguiente forma. Primero, se envían los códigos de hash a través de la red (un total de $\frac{N}{|B_{hc}|}$ *buffers*). Entonces responde con los *bitmaps* que son enviados al *sending end* (un total de $\frac{N}{|B_{be}|}$ *buffers*). Finalmente,

aquellas tuplas que no son filtradas se envían al *receiving end* ($\frac{N(S_{bf})}{|B_d|}$). Durante este proceso, se enviarán un número determinado de *buffers* de control dependiendo de la capacidad C del *receiving end*:

$$t_{comm_{RBFR}} = N\left(\left(\frac{1}{|B_{hc}|} + \frac{1}{|B_{be}|} + \frac{(S_{bf})}{|B_d|}\right) B_{d_s} + \frac{(S_{bf})}{|B_d|C} B_{c_s}\right)\lambda \tag{A.8}$$

Para RBF$_B$, el coste de la comuncicación reside en la tuplas no filtradas, pues ni los códigos de hash ni los *bitmaps* han de ser enviados. También, un número determinado de *buffers* de control será enviado en le siguiente caso:

$$t_{comm_{RBFB}} = N\left(\frac{1}{|B_d|} B_{d_s} + \frac{1}{|B_d|C} B_{c_s}\right)(S_{bf})\lambda \tag{A.9}$$

B

MODELO ANALÍTICO PARA EL STAR HASH JOIN

A continuación presentamos un modelo analítico exclusivamente orientado a analizar las técnicas *bitmap join* (BJ), *Multi Hierarchical Clustering* MHC, y *Star Hash Join* (SHJ) durante la ejecución del join estrella en arquitecturas CLUMP.

El modelo analítico lo formulamos bajo las condiciones de un esquema *Star Hash Join* presentado en el Capítulo 5 Sección 5.2. El modelo también asume que: (i) las tablas de dimensión están restringidas por un sólo atributo, (ii) los datos tienen una distribución uniforme, (iii) no hay correlación entre los valores de distintos atributos, y (iv) las claves de join tienen valores únicos en sus respectivas tablas de dimensión.

Dadas las premisas del modelo, para cualquier nodo $j : j = 0..k-1$ y tabla de dimensión $i : i = 0..n-1$, los *bit filters* creados durante la ejecución de la técnica SHJ, $BF_{i_0}, BF_{i_1}, ..., BF_{i_{m-1}}$, tienen todos la misma selectividad (Sbf): $Sbf_{i_0} = Sbf_{i_1} = .. = Sbf_{i_{k-2}} = Sbf_{i_{k-1}}$.

Como se explicó en el Capítulo 2 Sección 2.2.2, MHC puede acceder a los datos, multidimensionalmente organizados, de la tabla de hechos de distintas maneras. El modelo asume que el método de acceso utilizado por MHC para acceder a estos datos es óptimo, y que la sobrecarga es imperceptible desde el punto de vista de la E/S y los recursos de memoria. Para BJ, modelamos la E/S y la comunicación de datos sin tener en cuenta la sobrecarga que los índices podrían causar. Lo hacemos de esta forma ya que la técnica BJ está orientada a indicarnos cual es el máximo beneficio teórico que podemos obtener durante la ejecución de un join estrella.

Con el fin de modelar el tráfico de datos en la red y la E/S, primero necesitamos averiguar las cardinalidades de salida de cada nodo del plan de ejecución después de aplicar cualquiera de las tres técnicas. Después de ello, modelamos la capa de comunicación y la E/S. Finalizaremos el modelo analizando los recursos de memoria empleados por la técnica SHJ.

Datos a procesar por los nodos del plan.

Cada técnica tiene un efecto diferente sobre la reducción de datos a procesar sobre la tabla de hechos y, en consecuencia, sobre las cardinalidades de salida de los nodos del plan de ejecución. Empezamos dando las siguientes definiciones:

- OC_{i_j} valor original para la cardinalidad de salida del join entre la i^{esima} tabla de dimensión y la tabla de hechos en un nodo j: $F_j \bowtie D_{i_j}$.

- $S_{D_{i_j}}$ selectividad de la i^{esima} tabla de dimensión en cualquier cluser j.
- NOC_{i_j} nueva cardinalidad de salida del join $D_{i_j} \bowtie F_j$
- NOC_{F_j} nueva cardinalidad de salida de la operación de *scan* sobre la tabla de hechos.
- n número de tablas de dimensión que hacen join con la tabla de hechos.

Calculamos NOC_{i_j} y NOC_{F_j} de forma diferente para cada técnica:

BJ El *bitmap join* sólo necesita la información relacionada con la selectividad de las tablas de dimensión. Basado en ello, calculamos las cardinalidades de salida para cualquier nodo j de la siguiente forma:

$$NOC_{F_j} = \|F_j\| \left(\prod_{d=0}^{n-1} S_{D_{d_j}} \right) \tag{B.1}$$

$$NOC_{i_j} = OC_{i_j} \times \left(\prod_{d=i} n - 1 S_{D_{d_j}} \right) \tag{B.2}$$

MHC En este caso necesitamos definir R como el conjunto de tablas de dimensión cuyos atributos por los que están restringidas en la consulta, coinciden con los atributos aplicados para multidimensionar físicamente la tabla de hechos en cada nodo. Entonces, definimos las nuevas cardinalidades para cualquier nodo j de salida como:

$$NOC_{F_j} = \|F_j\| \left(1 - \prod_{d \in R} S_{D_{d_j}} \right) \tag{B.3}$$

$$NOC_{i_j} = OC_{i_j} \times \left(1 - \prod_{d > i : d \in R} S_{D_{d_j}} \right) \tag{B.4}$$

SHJ Esta técnica depende del factor de selectividad de los *bit filters* creados a partir de cada tabla de dimensión en cualquier nodo. Formulamos la nueva cardinalidad de salida para cualquier nodo j como:

$$NOC_{F_j} = \|F_j\| \left(1 - \prod_{d=0}^{n-1} Sbf_{d_j} \right) \tag{B.5}$$

$$NOC_{i_j} = OC_{i_j} \times \left(1 - \prod_{d=i}^{n-1} Sbf_{d_j} \right) \tag{B.6}$$

Comunicación y E/S

Empezamos dando las siguientes definiciones:

M páginas de memoria disponibles para la operación de cada operador de join.

k número de nodos en el sistema.

rec_size_F tamaño en bytes de cada tupla almacenada en la tabla de hechos $\frac{|F_j|}{\|F_j\|}$.

rec_size_{scan} tamaño en bytes de cada tupla proyectada por el *scan* sobre la tabla de hechos.

rec_size_{join} tamaño en bytes de la tupla resultante del join $F_j \bowtie D_{i_j}$.

SMP_p grado de paralelismo en cada nodo SMP.

λ factor de fuga para contar el añadido que supone el espacio extra necesario para gestionar los datos durante la ejecución de la fase de *build*.

ν factor de fuga para contar el añadido que supone el espacio extra para gestionar una tupla.

Contabilizamos, en bytes, el tráfico de datos en la capa de comunicación producido por cualquier técnica de la siguiente forma:

$$\sum_{d=1}^{n-1} NOC_{d_j} \times \frac{k-1}{k} \times rec_size_{join}\nu \tag{B.7}$$

Exceptuando el join con la primera tabla de dimensión, todos los hash joins añaden tráfico de datos a la capa de comunicación. $\frac{k-1}{k}$ es la fracción de datos comunicada sobre la cardinalidad de salida del operador de join.

Para la E/S, precisamos separar la que es producida por el *scan* sobre la tabla de hechos, y la que es producida por los operadores de join.

- Tabla de hechos. SHJ no afecta a la E/S salida sobre la tabla de hechos ya que el filtrado se produce una vez se han leído los datos. Así pues la E/S sobre la tabla de hechos para SHJ es : $\|F\| \times rec_size_F\nu$.

  Por otro lado, MHC y BJ reducen la E/S sobre la tabla de hechos. El número de tuplas leídas por el *scan* sobre la tabla de hechos es el mismo para las dos técnicas NOC_{F_j}. Así pues, calculamos, en bytes, la cantidad de E/S producida por el *scan* sobre la tabla de hechos para cualquier nodo j como:

$$E/S_{Fact} = NOC_{F_j} \times rec_size_F\nu \tag{B.8}$$

  Teniendo en cuenta que los nodo tienen una configuración SMP, y asumiendo que los datos están estratégicamente colocalizados en discos paralelos dentro de cada nodo, entonces el *scan* escala acorde al grado de paralelismo dentro de cada nodo: $\frac{E/S_{Fact}}{SMP_p}$.

- Operadores de join. La E/S producida por los operadores de join depende de la cantidad de datos volcada a disco durante la ejecución de fase de *build*. Definimos μ_i como la fracción

de la tabla tabla de dimensión i volcada a disco durante la ejecución de la operación de join. Calculamos μ_i como sigue:

$$\begin{cases} 1 - \frac{|M| - SMP_p}{|D_{i_j} \times S_{D_{i_j}}|\lambda}, & |D_{i_j} \times S_{D_{i_j}}|\lambda > |M|; \\ 0, & \text{de otra forma;} \end{cases}$$

Tomando como modelo el análisis del Hybrid Hash Join paralelo propuesto en [54] y explicado en el Capítulo 2, cada proceso escribe en una página distinta de memoria con tal de evitar contención: $|M| - SMP_p$. Entonces, las páginas de memoria ocupadas por los datos seleccionados de la tabla de dimensión se calculan como: $|D_{i_j} \times S_{D_{i_j}}|\lambda$. Así pues, calculamos la cantidad de datos leídos y escritos por cada nodo durante la ejecución de un join como:

$$E/S_{join} = 2 \times [|D_{j_0}|\mu_0 + NOC_{F_j}\mu_0(rec_size_{scan}\nu) +$$

$$\sum_{d=1}^{n-1} |D_{d_j}|\mu_i + (NOC_{d_j}\mu_d) \times (rec_size_{join}\nu)] \tag{B.9}$$

$|D_{d_j}|\mu_d$ para $d = 0..n-1$ es la E/S provocada por la fracción de datos de las tablas de dimensión volcada a disco por cada operador de join. $NOC_{F_j}\mu_0$ es el número de tuplas volcadas a disco por el primer operador de join, y $NOC_{i_j}\mu_i$ el número de tuplas volcadas a disco por el resto. Al igual que ocurría con la E/S de la tabla de hechos, como cada nodo tiene una configuración SMP y tiene múltiples discos, asumimos que los procesos, en cada nodo, escriben y leen los datos de las particiones volcadas a disco en paralelo. De este modo, se paraleliza la E/S internamente en cada nodo: $\frac{E/S_{join}}{SMP_p}$.

Recursos de memoria SHJ

Asociamos la memoria requerida por SHJ a la memoria requerida para almacenar los *bit filters*. No contabilizamos la memoria utilizada por el resto de operadores del plan de ejecución, pues esa memoria es intrínseca en la ejecución del join estrella para cualquier técnica.

Para todos las operaciones de join no colocalizadas, SHJ necesita almacenar los *bit filter* de cada nodo computacional involucrados en el join paralelo. Siendo m_i (ver Capítulo 2, Sección 2.4.1) las posiciones requeridas por el *bit filter* de la tabla de i^{esima} dimensión para cualquier nodo j, calculamos la memoria, en bytes, ocupada por ese *bit filter* como $Mbf_i = \frac{m_i}{8}$. Entonces, calculamos la cantidad de memoria M requerida por todo el sistema para almacenar los *bit filters* como:

$$M = Mbf_0 + \left[\sum_{d=1}^{n-1}(k) \times (Mbf_{d_j})\right] \tag{B.10}$$

El primer join entre $D_{0_j} \bowtie F_j$ es colocalizado, y por lo tanto sólo se necesita espacio para un *bit filter* por nodo Mbf_0.

C

MODELO ANALÍTICO PARA EL HYBRID HASH JOIN

En esta Sección definimos un modelo analítico para los tres algoritmos explicados en el Capítulo 6: *Basic*, *Non Memory-Conscious* (NMC) y *Memory-Conscious* (MC). El modelo que proponemos es una adapatación del modelo propuesto en [36, 54]. Nosotros centramos nuestro análisis en la fase de join del algoritmo *Hybrid Hash Join* (HHJ), y añadimos más complejidad considerando la posible aparición de bucket overflows, y la presencia de distribuciones de datos no uniformes. Así pues, los buckets procesados durante la fase de join del algoritmo HHJ pueden tener tamaños distintos, lo cual nos obliga a modelar cada bucket de forma independiente.

Denotaremos por $||X||, |X|$ la cardinalidad y el tamaño de X respectivamente. Empezamos dando la definición de algunas variables:

R_i, S_i , i^{esimo} par de buckets i para las relaciones de *build* R y *probe* S.

p_i , número de procesos asignados al par de buckets $\langle S_i, R_i \rangle$.

M_i , memoria disponible por los procesos p_i asignados al bucket $\langle S_i, R_i \rangle$.

BO_i , número de bucket overflows resultado de realizar el join entre un par de buckets dados $\langle S_i, R_i \rangle$, $\frac{|R_i| \times F}{|M_i|}$.

IO_{time} , tiempo necesario para realizar un acceso secuencial a disco.

Res_i , resultado del join entre el par de buckets $\langle S_i, R_i \rangle$.

ε_i , contención de memoria cuando se inserta una tupla del bucket R_i a través de la tabla de hash.

t_{hash} , tiempo de computación de la función de hash para una calve dada.

t_{insert} , tiempo necesario para insertar una tupla de R_i en la tabla de hash. Incluye el tiempo de lock y el tiempo de unlock de una página cuando hay varios procesos insertando tuplas en la tabla de hash.

t_{find_match} , tiempo necesario para, dada una tupla del bucket S_i, comparar esa tupla con todas las tuplas del bucket R_i mapeadas en la misma entrada de la tabla de hash.

t_{build_tuple} , tiempo necesario para construir una tupla resultado.

Modelamos la contención de memoria de la misma forma descrita en [54]. Podemos aproximar el número de inserciones concurrentes en una misma página de una tabla de hash a través del teorema de Yao [84]:

$$\gamma_i = \prod_{j=1..p_i} \frac{||R_i|| \times \left(1 - \frac{1}{|M_i|}\right) - j + 1}{||R_i|| - j + 1} \tag{C.1}$$

De este modo, la contención de memoria para un bucket dado se expresa como $\varepsilon_i = \frac{p_i}{\gamma_i}$, siendo una función que depende de R_i, M_i y p_i. También denotamos ε_i como $C(||R_i||, |M_i|, p_i)$. Estas variables pueden tomar distintos valores dependiendo del algoritmo modelado. Por ejemplo, el algoritmo NMC siempre tiene $p_i = 1$, de modo que está libre de contención de memoria, $\varepsilon_i = 1$.

Definimos el tiempo de E/S (t_{IO_i}) empleado durante la ejecución el join entre el par de buckets $\langle S_i, R_i \rangle$ como:

$$t_{IO_i} = IO_{time} \times (|R_i| + |S_i| + |S_i| \times BO_i) \tag{C.2}$$

La mínima E/S es leer los buckets R_i y S_i: $|R_i| + |S_i|$. Entonces si hay bucket overflows, tenemos que releer el bucket S_i un total de BO_i veces: $|S_i| \times BO_i$.

Calculamos el tiempo de cpu (t_{CPU_i}) empleado para la ejecución del join entre el par de buckets $\langle S_i, R_i \rangle$ como:

$$\begin{aligned} t_{CPU_i} = \tfrac{1}{p_i} \times (||R_i|| \times (t_{hash} + t_{insert} \times \varepsilon_i)) \\ +((||S_i|| \times BO_i) + ||S_i||) \times (t_{hash} + t_{find_match}) + ||Res_i|| \times t_{build_tuple} \end{aligned} \tag{C.3}$$

Tenemos que realizar el hash y la inserción de todas las tuplas de R_i: $|R_i| \times (t_{hash} + t_{insert} \times \varepsilon_i)$. Luego, se han de chequear las tuplas de S_i en la tabla de hash: $|S_i| \times t_{hash} + t_{find_match}$. Si hay bucket overflows, entonces, hay que chequear cada tupla un total de BO_i veces más: $|S_i| \times BO_i$. Finalmente, tenemos que construir el resultado: $|Res_i| \times t_{build_tuple}$.

Siendo $\langle S_i, R_i \rangle_i : i = 1..B$ el grupo de pares de bucket a procesar durante la fase de join del algoritmo HHJ, entonces, calculamos el total de cpu y E/S a realizar como:

$$t_{CPU} = \sum_{i=1..B} t_{CPU_i} \tag{C.4}$$

$$t_{IO} = \sum_{i=1..B} t_{IO_i} \tag{C.5}$$

Este modelo analítico aporta más precisión a las estimaciones de costes de E/S y tiempo de cpu para la fase de join del algoritmo HHJ. Especialmente bajo los efectos de distribuciones de datos no uniformes. Esto se debe a que el modelo da el coste de cada bucket de forma separada, de modo que, es posible determinar el coste de procesado de buckets con sesgo durante la fase de join del algoritmo. Esto es beneficioso para el optimizador del SGBD, porque nuestro modelo afronta distribuciones no uniformes, dando estimaciones precisas de coste para la fase de optimización.

D

ARTÍCULOS PUBLICADOS DURANTE EL DESARROLLO DE ESTA TESIS

1. Josep Aguilar-Saborit, Victor Muntes-Mulero and P. Trancoso and Josep-L. Larriba-Pey. ***Dynamic Count Filters***. SIGMOD Record, (35)1:26–32, March 2006.

2. Josep Aguilar-Saborit, Victor Muntes-Mulero, Adriana Zubiri, Calisto Zuzarte and Josep-L. Larriba-Pey. ***Dynamic out of core join processing in symmetric multiprocessors***. PDP'06: 14th Euromicro International Conference on Parallel, Distributed and Network based Processing, France, 2006.

3. Josep Aguilar-Saborit, Victor Muntes-Mulero, Calisto Zuzarte and Josep-L. Larriba-Pey. ***Ad-hoc star hash join processing in Clusters Architectures***. DAWAK'05: 7th International Conference in Data Warehousing and Knowledge Discovery, 200–209, Copenhagen, Denmark, 2005.

4. Josep Aguilar-Saborit, Victor Muntes-Mulero, Josep-L. Larriba-Pey, Calisto Zuzarte and Hebert Pereyra. ***On the use of bit filters in shared nothing partitioned systems***. IWIA'05: International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems, 29–37, Oahu, Hawaii, 2005.

5. Josep Aguilar-Saborit, Victor Muntes-Mulero and Josep-Lluis Larriba-Pey. ***Pushing Down Bit Filters in the Pipelined Execution of Large Queries***. Euro-Par'03: Parallel Processing, 9th International Euro-Par Conference, 328–337, Austria, 2003.

6. Josep Aguilar-Saborit, Victor Muntes-Mulero and Josep-Lluis Larriba-Pey. ***Pushing Down Bit Filters in the Pipelined Execution of Large Queries***. XIII Jornadas de Paralelismo, 389–394. Universitat de Lleida/Universitat Autonoma de Barcelona, 2002.

7. Josep Aguilar-Saborit, Victor Muntes-Mulero, Josep-L. Larriba-Pey, Calisto Zuzarte and Hebert Pereyra. ***A data traffic memory efficient protocol for the use of bit filter in shared nothing partitioned systems***. XV Jornadas de Paralelismo, 444–449. Almeria, Spain, 2003.

8. Josep Aguilar-Saborit, Victor Muntes-Mulero, Adriana Zubiri, Calisto Zuzarte and Josep-L. Larriba-Pey. ***Dynamic out of core join processing in symmetric multiprocessors***. Technical

Report. Computer Architecure Department. Universitat Politecnica de Catalunya UPC-DAC-2005-6.

9. Josep Aguilar-Saborit, Victor Muntes-Mulero, Calisto Zuzarte and Josep-L. Larriba-Pey. ***Ad-hoc star hash join processing in Clusters Architectures***. Technical Report. Computer Architecure Department. Universitat Politecnica de Catalunya UPC-DAC-RR-GEN-2005-4.

10. Josep Aguilar-Saborit, Victor Muntes-Mulero, Josep-L. Larriba-Pey, Calisto Zuzarte and Hebert Pereyra. ***A data traffic memory efficient protocol for the use of bit filter in shared nothing partitioned systems***. Technical Report. Computer Architecure Department. Universitat Politecnica de Catalunya. UPC-DAC-2004-22.