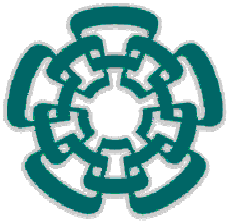


CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Fragmentación Vertical Dinámica de Bases de Datos Multimedia Usando Reglas Activas

TESIS

Que presenta

M. en C. Lisbeth Rodríguez Mazahua

Para obtener el grado de

Doctora en Ciencias en Computación

Directora de Tesis:

Dra. Xiaoou Li Zhang

México, D.F. **Diciembre 2012**

Índice general

Índice de figuras

Índice de tablas

Glosario de siglas

Sigla	Definición
AAM	Attribute Affinity Matrix
	Matriz de Afinidad de Atributos
AAT	Attribute Affinity Table
	Tabla de Afinidad de Atributos
ASM	Attribute Support Matrix
	Matriz de Soporte de Atributos
AT	Attributes Table
	Tabla de Atributos
ATM	Asynchronous Transfer Mode
	Modo de Transferencia Asíncrono
AUM	Attribute Usage Matrix
	Matriz de Uso de Atributos
AUT	Attribute Usage Table
	Tabla de Uso de Atributos
AVP	Adaptable Vertical Partitioning
	Fragmentación Vertical Adaptable
BDOO	Base de Datos Orientada a Objetos
BEA	Bond Energy Algorithm
	Algoritmo de Enlace de Energía
CA	Cluster Analyzer
	Analizador de Grupos

CAT	Clustered Affinity Table
	Tabla de Afinidad Agrupada
CBPA	Connection Based Partitioning Algorithm
	Algoritmo de Fragmentación Basado en Conexiones
DBA	DataBase Administrator
	Administrador de la Base de Datos
DBMS	DataBase Management System
	Sistema Gestor de Base de Datos
DDL	Data Definition Language
	Lenguaje de Definición de Datos
DRA	Decreased Remote Attribute accessed
	Cantidad Reducida de Atributos Remotos Accedidos
DYMOND	DYnamic Multimedia ON line Distribution
DYVEP	DYnamic VErtical Partitioning
ECA	Evento-Condición-Acción
EFV	Esquema de Fragmentación Vertical
FQ	Frequently-asked Queries
	Consultas Frecuentes
FVD	Fragmentación Vertical Dinámica
FVE	Fragmentación Vertical Estática
GCS	Global Conceptual Scheme
	Esquema Conceptual Global
IIA	Increased Irrelevant Attribute Accessed
	Cantidad Incrementada de Atributos Irrelevantes Accedidos
IAAC	Irrelevant Attribute Access Cost
	Costo de Acceso a Atributos Irrelevantes
ILP	Inductive Logic Programming
IMS	Information Management System
LCS	Local Conceptual Scheme
	Esquema Conceptual Local

MAN	Metropolitan Area Network
	Red de Área Metropolitana
MAUM	Method-Attribute Usage Matrix
	Matriz de Uso de Métodos y Atributos
MAVP	Multimedia Adaptable Vertical Partitioning
MMDB	MultiMedia DataBase
	Base de Datos MultiMedia
MMUM	Method-Method Usage Matrix
	Matriz de Uso de Métodos y Métodos
OHPE	Object Horizontal Partition Evaluator
	Evaluador de Particiones Horizontales de Objetos
OODBMS	Object-Oriented Database Management System
	Sistema Gestor de Base de Datos Orientado a Objetos
PE	Partition Evaluator
	Evaluador de Particiones
PEOO	Partition Evaluator for Distributed Object-Oriented Databases
	Evaluador de Particiones Orientado a Objetos
PT	Partition Tree
	Árbol de Partición
QT	Queries Table
	Tabla de Consultas
SC	Statistic Collector
	Recolector de Estadísticas
SQL	Structured Query Language
SVM	Support Value Matrix
	Matriz de Valores de Soporte
SVP	Support-Based Vertical Partitioning
TAUM	Transaction Attribute Usage Matrix
	Matriz de Uso de Transacciones y Atributos

TC	Transportation Cost
	Costo de Transporte
TMUM	Transaction-Method Usage Matrix
	Matriz de Uso de Transacciones y Métodos
TSP	Traveling Salesman Problem
	Problema del Agente Viajero
URC	User Reference Cluster
	Grupo de Referencia de Usuario
VPA	Vertical Partitioning Algorithm
	Algoritmo de Fragmentación Vertical

Agradecimientos

A DIOS por darme la vida, la salud, la fuerza y el amor, porque todo lo que tengo y lo que soy es gracias a él.

A mi padre Juan Francisco Rodriguez Antonio por haberme inculcado el gusto por el estudio desde muy pequeña, por ayudarme en mis problemas y por ser un ejemplo a seguir.

A mi madre Celerina Mazahua Tzitzihua por el apoyo incondicional, porque cuando siento que ya no puedo más, siempre me brinda una palabra de aliento para continuar en este arduo camino.

A mi hermana Nidia Rodriguez Mazahua por haberme motivado a seguir adelante.

A mi asesora, la Dra. Xiaoou Li Zhang, por la confianza, paciencia y gentileza brindada y por sus sabios consejos que me ayudaron mucho.

A los revisores por sus valiosos comentarios para el mejoramiento de esta tesis Dr. Debrup Chakraborty, Dr. Amilcar Meneses Viveros, Dr. Iván López Arévalo, Dra. Lorena Chavarría Báez y Dra. Maricela Claudia Bravo Contreras.

A mi amigo Asdrúbal López Chau por todo el apoyo brindado.

A Sofia Reza y Felipa Rosas por su ayuda administrativa.

A CONACYT por la beca para estudiar el doctorado.

Al CINVESTAV por la beca de obtención de grado.

Resumen

Una de las actividades más importantes del diseño de bases de datos distribuidas es la fragmentación, esta consiste en dividir una relación o tabla en un conjunto de relaciones (esquema de fragmentación), que contienen subconjuntos de tuplas (fragmentación horizontal), atributos (fragmentación vertical) o tuplas y atributos (fragmentación híbrida) para reducir el número de accesos a disco, así como el transporte de datos necesarios para realizar consultas y de esta forma reducir el tiempo de respuesta de dichas consultas.

La fragmentación vertical es una técnica de diseño de bases de datos distribuidas ampliamente utilizada en las bases de datos tradicionales (relacionales y orientadas a objetos que sólo contienen atributos textuales y numéricos) para reducir el número de atributos irrelevantes accedidos por las consultas. Actualmente, debido a la popularidad de las aplicaciones multimedia en Internet, surge la necesidad de utilizar estrategias de fragmentación en las bases de datos multimedia para aprovechar sus ventajas potenciales con respecto a la optimización de consultas. En las bases de datos multimedia los atributos tienden a ser objetos multimedia muy grandes, y por lo tanto, la reducción del número de accesos a objetos no utilizados por las consultas (irrelevantes) implicaría un ahorro considerable en el costo de ejecución de las consultas.

Sin embargo, la utilización de las técnicas de fragmentación vertical en bases de datos multimedia implica dos problemas: 1) los algoritmos de fragmentación vertical sólo toman en cuenta datos alfanuméricos y 2) el proceso de fragmentación se realiza de manera estática. Para resolver estos problemas proponemos un sistema basado en reglas activas (o

sistema activo) llamado DYMOND (DYnamic Multimedia ON line Distribution) que realiza una fragmentación vertical dinámica en bases de datos multimedia.

Nuestras contribuciones son las siguientes:

1. Creamos un método de fragmentación vertical para bases de datos relacionales, el cual se basa en soporte, una medida utilizada en minería de reglas de asociación. La principal ventaja de nuestro método es que obtiene automáticamente el valor del umbral de soporte mínimo y utilizando dicho umbral encuentra el esquema de fragmentación vertical óptimo, mientras que otros métodos relacionados prueban diferentes umbrales, con los que obtienen distintos esquemas de fragmentación vertical y después utilizan un modelo de costo para encontrar el óptimo.

2. Desarrollamos un algoritmo de fragmentación vertical para bases de datos multimedia, llamado MAVP (Multimedia Adaptable Vertical Partitioning). La principal ventaja de MAVP es que toma en cuenta el tamaño de los atributos para obtener fragmentos que minimizan la cantidad de datos irrelevantes accedidos y datos transportados por las consultas.

3. Presentamos un modelo de costo para evaluar esquemas de fragmentación vertical en bases de datos multimedia distribuidas. Algunos modelos de costo existentes sólo miden el costo de acceso a disco (E/S), pero no consideran el costo de transporte que es un factor que domina el costo de las consultas en bases de datos multimedia distribuidas. Los modelos que si miden el costo de transporte no toman en cuenta el tamaño de los atributos. Nuestro modelo de costo utiliza el tamaño de los atributos para obtener el costo de acceso a atributos irrelevantes, así como el costo de transporte de las consultas en una base de datos multimedia distribuida.

4. Proponemos un sistema activo de fragmentación vertical dinámica para bases de datos relacionales llamado DYVEP (DYnamic VErtical Partitioning). Usando reglas activas DYVEP puede monitorizar consultas ejecutadas en la base de datos con el fin de acumular la información necesaria para desarrollar la fragmentación vertical y

disparar automáticamente el proceso de fragmentación si se determina que el esquema de fragmentación vertical se ha vuelto inadecuado debido a cambios en los patrones de acceso o en el esquema de la base de datos.

5. Extendemos DYVEP para utilizarlo en bases de datos multimedia (DYMOND). DYMOND tiene cuatro módulos basados en reglas activas: 1) un *Recolector de Estadísticas* que almacena, filtra y analiza información relacionada con las consultas que se realizan en la base de datos, como la frecuencia de acceso de las consultas y los objetos multimedia que estas acceden, 2) un *Evaluador de Desempeño*, que obtiene el costo del esquema de fragmentación actual basado en el modelo de costo propuesto y si el costo es mayor que un umbral, entonces dispara el algoritmo de fragmentación vertical, 3) *MAVP* que genera un esquema de fragmentación vertical que minimiza el costo de ejecución de las consultas, y 4) un *Generador de Fragmentos*, que materializa el esquema de fragmentación encontrado por MAVP.

Abstract

One of the most important activities of distributed database design is partitioning, this consists of dividing a relation or table into two or more relations (partitioning scheme) which contain subsets of either tuples (horizontal partitioning), attributes (vertical partitioning), or tuples and attributes (hybrid fragmentation) to reduce both the number of disk accesses and the remote data transported needed to answer a query, and in this way, reducing the response time of such queries.

Vertical partitioning is a distributed database design technique widely employed in traditional databases (relational and object-oriented databases that only manage numeric and textual attributes) to reduce the number of irrelevant attributes accessed by the queries. Currently, due to the popularity of multimedia applications on the Internet, the need of using partitioning strategies in multimedia databases has arisen in order to use their potential advantages with regard to query optimization. In multimedia databases, the attributes tend to be of very large multimedia objects. Therefore, the reduction in the number of accesses to objects which are not used by the queries (irrelevant) would imply a considerable cost saving in the query execution.

Nevertheless, the use of vertical partitioning techniques in multimedia databases implies two problems: 1) current vertical partitioning algorithms only take into account alphanumeric data, and 2) the partitioning process is carried out in a static way. In order to solve these problems, we propose an active rule-based system (or active system) called DYMOND (DYnamic Multimedia ON line Distribution) which performs a dynamic vertical

partitioning in multimedia databases.

Our contributions are the following:

1. We create a vertical partitioning method for relational databases, which is based on support, a measure used in association rule mining. The main advantage of our method is that it automatically obtains the value of the threshold of minimum support, and according to this threshold, it finds the optimal vertical partitioning scheme, while other related methods probe different thresholds to get different vertical partitioning schemes, and then they use a cost model to find the optimal.

2. We develop a vertical partitioning algorithm for multimedia databases, called MAVP (Multimedia Adaptable Vertical Partitioning), The main advantage of MAVP is that it takes into account the size of the attributes to obtain fragments that minimizes the amount of both irrelevant data accessed and data transported by the queries.

3. We present a cost model to evaluate vertical partitioning schemes in distributed multimedia databases. Some existing cost models only measure the number of disk accesses (I/O) to query processing, but they do not consider the transportation cost, which dominates query costs in distributed multimedia databases. The models which measure the transportation cost do not take into account the size of the attributes. Our cost model uses the size of the attributes to obtain both the cost of accessing irrelevant attributes and the transportation cost of the queries in a distributed multimedia database.

4. We propose an active system for dynamic vertical partitioning of relational databases, called DYVEP (DYnamic VErtical Partitioning). Using active rules, DYVEP can monitor queries run against the database in order to accumulate the necessary information to develop the vertical partitioning, and automatically trigger the partitioning process if it is determined that the vertical partitioning scheme has become inadequate due to changes in access patterns or in the database scheme.

5. We extend DYVEP to use it in multimedia databases (DYMOND). DYMOND has four active rule-based modules: 1) The *Statistics Collector* stores, filters and analyzes information related to the database queries, like the access frequency of such queries and the multimedia objects accessed by them, 2) The *Performance Evaluator* calculates the cost of the current vertical partitioning scheme using the proposed cost model, and if the cost is greater than a threshold, then it triggers the vertical partitioning algorithm, 3) MAVP generates a vertical partitioning scheme, which minimizes the execution cost of the queries, and 4) the *Partitioning Generator* materializes the partitioning scheme found by MAVP.

Capítulo 1

Introducción

La constante evolución de la capacidad de almacenamiento de los dispositivos de memoria secundaria y en la velocidad de los dispositivos de procesamiento generó la necesidad de tener herramientas diseñadas específicamente para el procesamiento de grandes bancos de información, por tal motivo en la década de 1960 surgieron los Sistemas Gestores de Bases de Datos (DBMSs, en inglés: DataBase Management Systems). Integrated Data Store e Information Management System (IMS) fueron los primeros sistemas de este tipo.

Un DBMS es una herramienta poderosa para crear y administrar grandes cantidades de datos eficientemente, así como para permitir que los datos persistan a través de largos periodos de tiempo de manera segura. Una base de datos es una colección de datos administrada por un DBMS [1]. Las bases de datos contienen información relevante para una organización [2].

Antes del surgimiento de los DBMSs, las organizaciones usaban sistemas de procesamiento de archivos para almacenar su información. Los principales inconvenientes de estos sistemas son que cada usuario define e implementa los archivos necesarios para una aplicación específica. Así que cualquier cambio a la estructura de un archivo puede requerir cambiar todos los programas que lo acceden. Además, distintos usuarios replican datos en diferentes archivos, resultando fácilmente en inconsistencias posteriores. En contraste, los DBMSs mantienen sólo un repositorio que se define una vez y entonces se accede por varios

usuarios [3].

Los programadores que usaban los primeros DBMSs estaban forzados a tratar muchos detalles de implementación de bajo nivel y tenían que codificar sus consultas de forma procedimental, además debían tener presente el rendimiento durante el diseño de sus programas lo que implicaba un gran esfuerzo. En 1970, surgieron los DBMSs relacionales [4], aunque su auge empezó en la década de 1980 cuando se presentó el lenguaje SQL (Structured Query Language), en ese tiempo llamado SEQUEL [5]. Los DBMSs relacionales fueron tan fáciles de usar que se consolidaron como las bases de datos dominantes, ya que ocultaban completamente los detalles de implementación al programador y presentaban formas no procedimentales de consultar los datos.

En los DBMSs relacionales, la información se organiza en tablas. Las filas de la tabla corresponden a registros, mientras que las columnas corresponden a atributos. SQL se usa para crear dichas tablas y para borrar, modificar y recuperar información de ellas. En la actualidad, los DBMSs relacionales son los más utilizados y se desarrollan por varias compañías de software. DB2 de IBM, SQLServer de Microsoft y Oracle son los líderes en el mercado. Otros DBMSs relacionales incluyen INGRES, PostgreSQL, MySQL y dBase.

La década de 1980 también fue testigo del trabajo inicial en el desarrollo de Sistemas Gestores de Bases de Datos Orientados a Objetos (OODBMSs, en inglés: Object-Oriented DataBase Management Systems) [2], estos sistemas combinan capacidades de las bases de datos (almacenamiento y búsqueda) y características del paradigma orientado a objetos (encapsulamiento, herencia e identidad de objetos), por lo tanto, los objetos se definen de acuerdo a ese paradigma, es decir, cada objeto contiene propiedades o atributos y utiliza métodos o funciones para manipular dichas propiedades.

Originalmente la mayoría de los datos manejados por aplicaciones de computadora eran datos textuales. Por lo tanto, las bases de datos tradicionales (relacionales y orientadas a objetos) fueron diseñadas y optimizadas para manejarlos [6]. Los avances en la tecnología de computadoras y comunicaciones de 1990 (principalmente el crecimiento explosivo del World Wide Web) permitieron que el acceso en línea a información multimedia,

como libros, periódicos, revistas, imágenes, videoclips y datos científicos, fuera posible y económico [7]. Al principio, estos datos se almacenaban en sistemas de archivos, este tipo de almacenamiento no supone ningún problema cuando el número de datos multimedia es pequeño, ya que las características de las bases de datos (consistencia, concurrencia, integridad, seguridad) suelen no ser importantes. Sin embargo, las características de las bases de datos se vuelven importantes cuando el número de datos multimedia es grande [2].

Como la necesidad de sistemas de información multimedia creció rápidamente en varias áreas como educación (bibliotecas digitales, entrenamiento, presentación y aprendizaje a distancia), cuidado de la salud (telemedicina, gestión de información de salud, sistemas de imágenes médicas), entretenimiento (video en demanda, bases de datos de música, TV interactiva), diseminación de la información (noticias en demanda, publicidad, transmisión de TV) e industria (comercio electrónico), la gestión adecuada de dicha información se volvió un punto central de investigación en la comunidad de base de datos [8].

Una base de datos multimedia es una colección de datos multimedia relacionados. Los tipos de datos multimedia comunes que se pueden encontrar en dicha base de datos incluyen: texto, gráficos, imágenes, animaciones, audio y video [9].

Un Sistema Gestor de Bases de Datos Multimedia (DBMS multimedia) debe soportar los diversos tipos de datos multimedia y facilitar actividades como: creación, almacenamiento, acceso, consulta y control de la base de datos multimedia [10]. Los problemas fundamentales en la gestión de bases de datos multimedia enfrentados por los investigadores son los siguientes [8]:

- Desarrollo de modelos para capturar los requerimientos de sincronización de datos multimedia.
- Desarrollo de modelos conceptuales para información multimedia, especialmente para imágenes, audio y video.
- Diseño de métodos poderosos para indexación, búsqueda, acceso y organización de datos multimedia.

- Diseño de lenguajes de consulta multimedia eficientes.
- Desarrollo de esquemas de agrupamiento y almacenamiento de datos para gestionar datos multimedia de tiempo real.
- Diseño y desarrollo de arquitecturas adecuadas y soporte del sistema operativo.
- Gestión de bases de datos multimedia distribuidas [8], [10].

Esta tesis doctoral se enfoca en el último problema.

1.1. Gestión de bases de datos multimedia distribuidas

El DBMS multimedia distribuido se refiere a una colección de varios sistemas gestores de base de datos multimedia (posiblemente) independientes, localizados en ubicaciones distintas, que pueden comunicar e intercambiar datos multimedia sobre una red. Los sistemas multimedia son usualmente distribuidos en el sentido de que una consulta multimedia simple con frecuencia involucra datos obtenidos de repositorios de información distribuidos. Este es normalmente el caso en ambientes multimedia colaborativos, donde múltiples usuarios en, posiblemente, ubicaciones físicas distintas manipulan y crean el mismo documento multimedia. Además, aspectos como problemas de almacenamiento y generación de datos puede también forzar a los diseñadores del sistema multimedia a colocar datos multimedia en distintas ubicaciones físicas [10].

Para soportar la información requerida en dichos ambientes colaborativos y distribuidos, un DBMS multimedia distribuido debe tratar con los problemas generales en bases de datos distribuidas, estos problemas son [11]:

- Diseño de la base de datos distribuida
- Gestión del directorio distribuido
- Procesamiento distribuido de consultas
- Control de concurrencia distribuida

- Fiabilidad del DBMS distribuido
- Replicación

A diferencia de las bases de datos tradicionales, la replicación de datos no se fomenta en un DBMS multimedia distribuido debido a los enormes volúmenes de datos. El modelo de cómputo cliente-servidor, en el que una aplicación servidor sirve múltiples aplicaciones cliente (con los clientes y el servidor residiendo en posiblemente diferentes máquinas) ha resultado adecuado para sistemas multimedia en general y DBMSs multimedia distribuidos en particular.

Esta tesis se enfoca en el problema de diseño de la base de datos distribuida en un DBMS multimedia. Específicamente en la fragmentación vertical que es una de las actividades principales del diseño de bases de datos distribuidas.

1.2. Diseño de bases de datos distribuidas

Una base de datos distribuida es un conjunto de bases de datos parcialmente independientes que (idealmente) comparten un esquema común y se comunican entre sí a través de una red de computadoras [2].

La investigación en bases de datos distribuidas inició en la década de 1970 (anteriormente sólo se consideraban sistemas de bases de datos centralizados, los cuales se ejecutan en un único sistema informático, sin interaccionar con ninguna otra computadora). En ese tiempo, la principal meta era soportar la gestión de datos distribuidos de grandes corporaciones y organizaciones que mantenían sus datos en diferentes oficinas. Los primeros sistemas para bases de datos distribuidas (R*, SDD-1 y Distributed INGRES) no fueron exitosos en el mercado debido a que la tecnología de comunicación no era lo suficientemente estable para enviar megabytes de datos requeridos por estos sistemas y los grandes negocios de alguna forma lograron sobrevivir sin la tecnología de base de datos distribuida sofisticada enviando cintas, diskettes o simplemente papeles, para intercambiar datos entre sus oficinas.

Actualmente, debido a los avances tecnológicos de hardware, software y estándares, el desarrollo de sistemas distribuidos de bases de datos es cada vez más factible y necesario. Para cumplir este requerimiento, la mayoría de los vendedores de sistemas de bases de datos relacionales ofrecen productos que soportan la distribución de la base de datos, por ejemplo IBM, Microsoft, Oracle [12].

La motivación para el desarrollo de bases de datos distribuidas puede describirse con las siguientes propiedades [13]:

- **Fiabilidad y disponibilidad**. Fiabilidad se refiere a la habilidad para tolerar fallos y disponibilidad a la probabilidad de que el sistema pueda utilizarse durante periodos de tiempo deseados. Mejorar la fiabilidad y la disponibilidad son ventajas potenciales de bases de datos distribuidas. Cuando se colocan copias de los datos en diferentes sitios, si un sitio que contiene una relación falla, es posible acceder a la relación en otro sitio, además aunque los datos sean inaccesibles, el sistema distribuido de bases de datos todavía puede ofrecer servicios limitados.

- **Razones de organización.** Debido a los avances en las técnicas de comunicación, para muchas organizaciones, especialmente organizaciones globales que son descentralizadas, es más adecuado implementar sistemas de información de una forma descentralizada.

- **Capacidad de expansión.** Es más fácil acomodar bases de datos de tamaño creciente en un ambiente distribuido agregando nuevas sucursales o almacenes relativamente autónomos, con un grado mínimo de impacto en el sistema existente.

- **Autonomía local**. Cada ubicación es capaz de mantener un grado de control sobre los datos que se almacenan localmente.

- **Nuevas aplicaciones.** Muchas nuevas aplicaciones se basan en tecnología de bases de datos distribuidas, incluyendo comercio electrónico, aprendizaje a distancia, bibliotecas digitales, etc.

- **Desempeño.** Esto se refiere a la habilidad para mejorar la ubicación de los datos para reducir el tiempo de respuesta de las consultas e incrementar el rendimiento, y a la vez reducir el tamaño de los datos que necesitan transportarse, así como reducir la competencia por servicios de CPU y E/S de disco.

- **Razones económicas.** Es mucho menos costoso instalar un sistema de varias computadoras pequeñas con el poder equivalente de una sola máquina grande debido a los avances en las estaciones de trabajo y en las computadoras personales.

El procesamiento distribuido en DBMSs es una forma eficiente de mejorar el desempeño de aplicaciones que manejan grandes cantidades de datos, un ejemplo de estas aplicaciones son las bases de datos multimedia, las cuales deben soportar objetos de gran tamaño como las imágenes, el sonido y el video. Esto se logra removiendo datos irrelevantes accedidos durante la ejecución de las consultas y reduciendo el intercambio de datos entre sitios, que son las principales metas del diseño de bases de datos distribuidas [11].

El diseño de bases de datos distribuidas es uno de los principales problemas de investigación cuya solución se supone que mejora el desempeño de la base de datos. Este problema involucra adquisición de datos, fragmentación de la base de datos, asignación y replicación de los fragmentos y optimización local. La fragmentación y la asignación son los elementos más importantes de la fase de diseño de la base de datos distribuida, ya que permiten reducir el costo de las consultas que se ejecutan en la base de datos.

La **fragmentación** es una técnica de diseño para dividir una base de datos en un conjunto de particiones o fragmentos de tal forma que la combinación de las particiones produzca la base de datos original sin ninguna pérdida o adición de información [13]. Esto reduce la cantidad de datos irrelevantes accedidos por las consultas, reduciendo el número de accesos a disco. El resultado del proceso de fragmentación es un conjunto de fragmentos definidos por un esquema de fragmentación.

La **fragmentación** puede ser vertical, horizontal o híbrida. La **fragmentación vertical** divide una tabla en subconjuntos de atributos, cada subconjunto forma un fragmento

vertical; la **fragmentación horizontal** divide una tabla en subconjuntos de tuplas, cada subconjunto forma un fragmento horizontal; y la **fragmentación híbrida** consiste en aplicar fragmentación vertical y horizontal simultáneamente en una tabla.

La **asignación** es el proceso de ubicar cada fragmento en un nodo de la red después de que la base de datos se ha fragmentado apropiadamente [11]. Cuando los datos son asignados, éstos pueden replicarse o mantenerse una sola copia. La replicación de los fragmentos mejora la fiabilidad y eficiencia de las consultas de sólo lectura. La intención de la asignación es minimizar el costo de la transferencia de datos y el número de mensajes necesarios para procesar un conjunto de consultas dado, para que el sistema funcione efectivamente y eficientemente [14]. Esta tesis se enfoca en el problema de la fragmentación.

Definición 1.1 *Si la relación R se fragmenta, R se divide en varios fragmentos* fr_1, fr_2, ..., fr_n. *Estos fragmentos contienen suficiente información, como para permitir la reconstrucción de la relación original R [2].*

El propósito del diseño de la fragmentación es determinar fragmentos no superpuestos que podrían ser la unidad apropiada de distribución. Una relación no es una unidad de distribución adecuada, así que es necesario fragmentarla por varias razones:

1. Generalmente las consultas acceden sólo a subconjuntos de una relación completa. Por lo tanto, la fragmentación puede reducir accesos a datos irrelevantes e incrementar la disponibilidad local de los datos.

2. Si las consultas que acceden a los datos de una relación dada residen en diferentes sitios, se pueden seguir dos alternativas con la relación completa como unidad de fragmentación: o la relación no se replica y se almacena en un sólo sitio, o se replica en todos o en algunos de los sitios donde residen las consultas. La primera alternativa resulta en un alto volumen innecesario de accesos a datos remotos. La segunda, por otro lado, tiene replicación innecesaria, que causa problemas en ejecutar actualizaciones y puede no ser deseable si el almacenamiento es limitado.

3. La descomposición de una relación en fragmentos, cada uno tratado como una unidad, permite que las transacciones se ejecuten concurrentemente. Además, la fragmentación de las relaciones típicamente resulta en la ejecución paralela de una consulta dividiéndola en un conjunto de subconsultas que operan sobre los fragmentos. Por lo tanto, la fragmentación incrementa el nivel de concurrencia y al mismo tiempo, el rendimiento del sistema.

Con la fragmentación surgen dificultades también. Si las consultas tienen requerimientos conflictivos que impiden la descomposición de la relación en fragmentos mutuamente exclusivos, aquellas aplicaciones que acceden a los datos ubicados en más de un fragmento pueden sufrir degradación de desempeño. Podría ser necesario, por ejemplo, recuperar datos de dos fragmentos y entonces hacer su reunión, la cual es muy costosa. Reducir las reuniones distribuidas es un problema fundamental de la fragmentación.

El segundo problema se relaciona con el control semántico de datos, específicamente con verificar la integridad. Como un resultado de la fragmentación, los atributos que participan en una dependencia pueden descomponerse en diferentes fragmentos que podrían asignarse a sitios diferentes. En este caso, incluso la tarea más simple de verificación de dependencias resultaría en la búsqueda de datos en distintos sitios.

La fragmentación puede optimizar la ejecución de las consultas en una base de datos porque generalmente las consultas sólo requieren subconjuntos de tuplas y/o atributos, cuando se fragmenta una tabla se reducen los accesos a datos irrelevantes (que no son requeridos por las consultas), esto permite un ahorro en el costo de ejecución de las consultas, mejorando el tiempo de respuesta considerablemente. La fragmentación se ha estudiado principalmente en bases de datos tradicionales (relacionales y orientadas a objetos que sólo consideran atributos textuales y numéricos). Recientemente, debido a la popularidad de las aplicaciones multimedia distribuidas (audio/video en demanda, televisión interactiva, juegos de video, comercio electrónico, educación en demanda, etc.), surgió la necesidad de aplicar técnicas de fragmentación en las bases de datos multimedia para mejorar la ejecución de las consultas [15], [16], [17], [18], [19].

1.3. Motivación

Las bases de datos multimedia deben lidiar con el uso creciente de enormes cantidades de datos multimedia [6]. La utilidad de estas bases de datos depende de qué tan rápido puedan recuperarse los datos, por lo tanto, los métodos de acceso y las técnicas de almacenamiento de datos eficientes son cada vez más críticos para mantener un tiempo de respuesta de consultas aceptable.

Las bases de datos multimedia contienen atributos de gran tamaño (audio, video, etc.) y muchas de las consultas no requieren acceso a todos los atributos. La fragmentación vertical es una técnica adecuada para emplearse en estas bases de datos, ya que permite reducir el acceso a atributos multimedia irrelevantes minimizando el tiempo de respuesta de las consultas considerablemente [20], [15], [21]. Esto se logra agrupando los atributos accedidos por las consultas como fragmentos verticales, por lo tanto, las consultas acceden sólo a los fragmentos que contienen los atributos que usan en lugar de acceder a la tabla completa.

Sin embargo, es necesario desarrollar nuevos métodos de fragmentación vertical (o extender los existentes) para utilizarlos en este tipo de base de datos porque los métodos existentes se enfocan en bases de datos tradicionales. Recientemente, se presentaron algunas propuestas para fragmentar bases de datos multimedia en la literatura pero sólo consideran fragmentación horizontal [16], [17], [18], [19]. El único trabajo que trata acerca de la aplicación de técnicas de fragmentación vertical en bases de datos multimedia para lograr ejecución de consultas eficiente es [15]. Una desventaja de dicho trabajo es que utiliza una fragmentación vertical estática, que consiste en crear un esquema de fragmentación vertical de acuerdo a la información de las consultas que acceden a la base de datos y sus frecuencias, así como del esquema de la base de datos. El administrador de la base de datos (DBA) obtiene dicha información en una etapa de análisis del sistema. Esto implica que cuando el sistema sufre suficientes cambios, el DBA debe realizar una nueva etapa de análisis del sistema para obtener un nuevo esquema de fragmentación. En caso contrario,

el esquema se degradará incrementando el tiempo de respuesta de las consultas.

Las bases de datos multimedia que soportan aplicaciones multimedia distribuidas son dinámicas, debido a que son accedidas por varios usuarios simultáneamente [22], [23], por lo tanto, las consultas y sus frecuencias tienden a cambiar con el tiempo y una técnica de fragmentación estática sería ineficiente en estas bases de datos [24]. Es mejor utilizar una técnica de fragmentación vertical dinámica que adapte automáticamente el esquema de fragmentación de acuerdo a los cambios que ocurran en la base de datos. La fragmentación vertical dinámica es un problema difícil porque se debe monitorizar continuamente la base de datos para detectar los cambios que ocurren, la relevancia de estos cambios necesita evaluarse y entonces debe iniciarse el proceso de fragmentación. Cualquier cambio en el esquema de fragmentación debe materializarse. La materialización se refiere a la reorganización de la base de datos para ajustarse al nuevo esquema de fragmentación.

Una forma efectiva de monitorizar la base de datos es usando reglas activas, ya que permiten realizar monitorización continua dentro del servidor de base de datos y tienen la habilidad de realizar acciones automáticamente con base en dicha monitorización, además debido a su simplicidad, pueden implementarse sin costos adicionales de memoria y CPU [25]. Una regla activa o regla ECA (Evento-Condición-Acción) define los eventos que deben monitorizarse, las condiciones que deben evaluarse y las acciones que deben tomarse, por ejemplo, la siguiente regla almacena la definición de una consulta en la tabla `queries` cada vez que se ejecuta una consulta y su tiempo de ejecución sea mayor a 100 segundos.

`ON` Query.Commit

`IF` Query.Duration $>$ 100

`THEN` Query.Persist (`queries`)

Es por estas razones que proponemos un sistema para la fragmentación vertical dinámica en bases de datos multimedia basado en reglas activas que permita obtener tiempos de respuesta de consultas aceptables en todo momento sin la intervención del DBA.

1.4. Contribuciones

Este trabajo de tesis tiene las siguientes contribuciones:

1. Un método de fragmentación vertical para bases de datos relacionales basado en la medida de soporte usada en minería de reglas de asociación. Su principal ventaja es que determina automáticamente el valor del umbral de desempeño mínimo para encontrar el esquema de fragmentación vertical óptimo.

2. Un algoritmo de fragmentación vertical para bases de datos multimedia que toma en cuenta el tamaño de los atributos y cuya función objetivo minimiza la cantidad de datos irrelevantes accedidos y la cantidad de datos transportados por las consultas.

3. Un modelo de costo para la evaluación de distintos esquemas de fragmentación vertical en bases de datos multimedia distribuidas que toma en cuenta el costo de acceso a atributos irrelevantes, así como el costo de transporte.

4. Un sistema de fragmentación vertical dinámica para bases de datos relacionales que utiliza reglas activas para acumular la información necesaria para desarrollar la fragmentación vertical y adaptar automáticamente el esquema de fragmentación vertical de acuerdo a los cambios en dicha información.

5. Un sistema de fragmentación vertical dinámica para bases de datos multimedia basado en reglas activas formado por los siguientes módulos:

   - Un recolector de estadísticas basado en reglas activas que almacena toda la información necesaria en el proceso de fragmentación vertical, analiza la información y en caso de que ocurran suficientes cambios dispara el evaluador de desempeño.
   - Un evaluador de desempeño que obtiene el costo del esquema de fragmentación actual basado en el modelo de costo propuesto y si el costo es mayor que un umbral, dispara el algoritmo de fragmentación vertical.

- Un algoritmo de fragmentación vertical que utiliza reglas activas para obtener un nuevo esquema de fragmentación con base en las estadísticas.
- Un generador de fragmentos que materializa el nuevo esquema de fragmentación, implementado los fragmentos como índices para proveer transparencia de fragmentación.

1.5. Organización de la tesis

El presente documento está estructurado de la siguiente manera: En el Capítulo 2 se presentan los conceptos básicos de las bases de datos distribuidas, así como el diseño de la distribución y el procesamiento de consultas. El Capítulo 3 se centra en los métodos de fragmentación estática y dinámica tanto en bases de datos tradicionales como en bases de datos multimedia, además describe el método propuesto de fragmentación vertical estática para bases de datos relacionales basado en soporte. En el Capítulo 4 se muestra la importancia de aplicar técnicas de fragmentación vertical en bases de datos multimedia, así como nuestra propuesta de fragmentación vertical estática para bases de datos multimedia llamada MAVP (Multimedia Adaptable Vertical Partitioning). En el Capítulo 5 se explica la motivación para desarrollar técnicas de fragmentación vertical dinámica en las bases de datos y nuestra propuesta de fragmentación vertical dinámica para bases de datos relacionales DYVEP (DYnamic VErtical Partitioning). El Capítulo 6 se enfoca en la arquitectura del sistema para fragmentación vertical dinámica de bases de datos multimedia propuesto, llamado DYMOND (DYnamic Multimedia ON-line Distribution). El Capítulo 7 aborda la implementación de DYMOND en el DBMS objeto-relacional PostgreSQL. En el Capítulo 8 se analiza la aplicación de DYMOND en un caso de estudio. Finalmente, en el Capítulo 9 se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Bases de datos distribuidas

La tecnología de bases de datos distribuidas surgió debido a la fusión de dos tecnologías: la de base de datos y la de comunicación de datos y de redes. La última ha crecido significativamente en términos de tecnologías alámbricas e inalámbricas -desde comunicación celular y satelital y redes de área metropolitana (MANs) hasta la estandarización de protocolos como Ethernet, TCP/IP y el Modo de Transferencia Asíncrono (ATM), así como la explosión del Internet, incluyendo el desarrollo del Internet-2. Mientras las primeras bases de datos se movieron hacia la centralización y resultaron en bases de datos gigantes monolíticas en los setentas y a principios de los ochentas, la tendencia cambió hacia más descentralización y autonomía de procesamiento a finales de los ochenta. Con los avances en la computación y el procesamiento distribuido de los sistemas operativos, la comunidad de investigación de bases de datos realizó trabajo considerable para atacar los problemas de distribución de datos, procesamiento de consultas y transacciones distribuidas, gestión de metadatos de las bases de datos distribuidas, entre otros, y desarrolló muchos prototipos de investigación. Sin embargo, un DBMS distribuido totalmente operativo que incluyera la funcionalidad y técnicas propuestas por la investigación en bases de datos distribuidas nunca surgió como un producto comercialmente viable. La mayoría de los fabricantes no centraron sus esfuerzos en el desarrollo de un DBMS distribuido "puro", sino en la generación de sistemas basados en la arquitectura cliente-servidor, o hacia desarrollar tecnologías para el acceso a

fuentes de datos distribuidos de manera heterogénea.

Sin embargo, las organizaciones han estado muy interesadas en la descentralización del procesamiento (al nivel del sistema) mientras se logra una integración de los recursos de información (al nivel lógico) en sus sistemas de bases de datos, aplicaciones y usuarios distribuidos geográficamente. Además de los avances en las comunicaciones, hay una aprobación general del esquema cliente-servidor para el desarrollo de aplicaciones, el cual asume muchos de los aspectos de bases de datos distribuidas [26].

2.1. Conceptos de bases de datos distribuidas

El principio fundamental detrás de la gestión de datos es la independencia de datos, que permite a las aplicaciones y usuarios compartir datos en un alto nivel conceptual mientras se ignoran los detalles de la implementación. Este principio se ha logrado por medio de sistemas de bases de datos que proporcionan capacidades avanzadas tales como gestión del esquema, lenguajes de consulta de alto nivel, control de acceso, procesamiento y optimización de consultas automático, transacciones, estructuras de datos para soportar objetos complejos, etc.

Una base de datos distribuida es una colección de multiples bases de datos, lógicamente interrelacionadas y distribuidas sobre una red de computadoras. Un sistema gestor de bases de datos distribuido (DBMS distribuido) es el conjunto de programas que permiten la administración de la base de datos distribuida y hace la distribución transparente a los usuarios. Un sistema distribuido de bases de datos se refiere a la base de datos distribuida y al DBMS distribuido. La transparencia de la distribución extiende el principio de la independencia de datos de tal forma que la distribución no es visible para los usuarios.

Estas definiciones asumen que cada sitio consiste lógicamente de una sola computadora independiente. Por lo tanto, cada sitio tiene la capacidad de ejecutar aplicaciones solo. Los sitios se interconcetan por medio de una red de computadoras con una conexion “suelta” (no comparten memoria ni disco) entre los sitios que operan independientemente. La aplicaciones pueden entonces someter consultas y transacciones e integrar los resultados. El

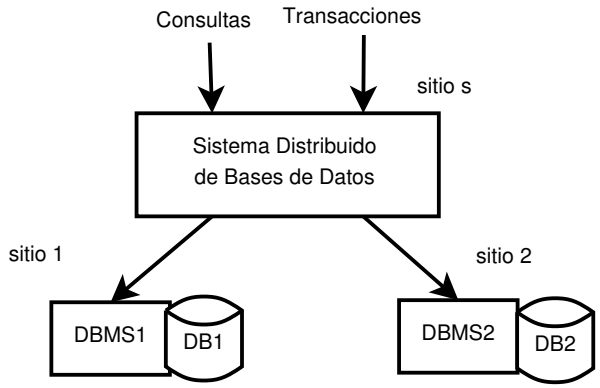


Figura 2.1: Ambiente de un sistema distribuido de bases de datos.

sistema distribuido de bases de datos puede ejecutarse en cualquier sitio no necesariamente distinto de los datos (esto es, puede ser en el sitio 1 o 2 en la Figura 2.1) [27].

2.1.1. Evolución de DBMSs distribuidos

Durante la década de 1970, algunas corporaciones implementaron DBMSs centralizados para satisfacer sus necesidades de información estructurada. La información estructurada por lo general se presenta como reportes formales emitidos regularmente en formato estándar. Esta información generada por lenguajes de programación procedimentales, es creada por especialistas que responden a peticiones de manera precisa. Por lo tanto, las necesidades de información estructurada estaban bien proporcionadas por los sistemas centralizados.

El uso de una base de datos centralizada requería que los datos se guardaran en un solo sitio central, por lo general un mainframe. El acceso a los datos se realizaba mediante terminales "tontas". El método centralizado funcionaba bien para llenar las necesidades de información estructurada de las corporaciones, pero ya no tenía capacidad cuando eventos que ocurrían con gran rapidez requerían un acceso rápido y no estructurado a la información usando consultas ad hoc para generar información en el acto.

Los DBMSs fundamentados en el modelo relacional proporcionaron el ambiente en el que se lograron satisfacer las necesidades de información no estructurada empleando con-

sultas ad hoc. Los usuarios finales lograron tener acceso a los datos cuando fuera necesario. Sin embargo, las primeras implementaciones del modelo relacional no dieron un rendimiento efectivo total cuando se compararon con los bien establecidos modelos de bases de datos jerárquicos o de red.

La década de 1980 fue un periodo my activo para el desarrollo de la tecnología de bases de datos relacionales distribuidas y todos los DBMSs comerciales actuales son distribuidos. La década de 1990 fue testigo del desarrollo y la madurez de la tecnología cliente-servidor y la introducción de la orientación a objetos -ambas como sistemas independientes y DBMSs objeto-relacionales [27].

Las últimas dos décadas vieron nacer una serie de decisivos cambios sociales y tecnológicos que afectaron el desarrollo y diseño de las bases de datos. Entre estos cambios se hallan:

- Las operaciones de negocios se descentralizaron.
- La competencia aumentó a nivel mundial.
- Las demandas de clientes y necesidades del mercado favorecieron un estilo de administración descentralizada.
- El rápido cambio tecnológico creó computadoras de bajo costo con potencia semejante a la de un mainframe, al igual que equipos inalámbricos portátiles que manejaban impresionantes funciones múltiples con telefonía celular y servicios de datos, así como redes cada vez más complejas y rápidas para conectarlos. En consecuencia, las corporaciones han adoptado cada día más tecnologías de red como plataforma para sus soluciones computarizadas.
- El gran número de aplicaciones basadas en los DBMSs y la necesidad de proteger las inversiones en el software de DBMSs centralizados hicieron atractiva la noción de compartir datos. Los diversos tipos de datos convergen en el mundo digital cada vez más. Por lo tanto, las aplicaciones manejan múltiples tipos de datos (voz, video, músi-

ca, imágenes, etc.) y a esos datos se tiene acceso desde varios lugares geográficamente dispersos.

Estos factores crearon un dinámico ambiente de negocios en el que las compañías tenían que responder rápidamente a presiones tecnológicas y de competitividad. Cuando las grandes unidades de negocios se reestructuraron para formar operaciones dispersas, menos complejas y de reacción rápida, se hicieron evidentes dos necesidades en bases de datos:

- El rápido acceso a los datos utilizando consultas ad hoc se hizo decisivo en el ambiente donde se requería una respuesta rápida para toma de decisiones.
- La descentralización de estructuras de administración, basada en la descentralización de unidades de negocios, convirtió en una necesidad las bases de datos de acceso múltiple descentralizado y ubicadas en diversos lugares.

Durante años recientes, los factores anteriores se afianzaron todavía más. No obstante, la forma en que se abordaron estuvo fuertemente influida por:

- La creciente aceptación de Internet como plataforma para acceso y distribución de datos. La World Wide Web es, en efecto, el depósito de los datos distribuidos.
- La revolución inalámbrica. El uso generalizado de equipos digitales inalámbricos, por ejemplo, teléfonos inteligentes como el iPhone y BlackBerry y las agendas electrónicas (PDA), han creado una alta demanda de acceso a los datos. Estos equipos tienen acceso a los datos desde lugares geográficamente dispersos y requieren variados intercambios de datos en gran cantidad de formatos (datos, voz, video, música, imágenes, etc.). Aunque el acceso a los datos distribuidos no necesariamente implica bases de datos distribuidas, es frecuente que el rendimiento y los requisitos de tolerancia a fallas hagan uso de técnicas de copia de datos similares a los que encontramos en bases de datos distribuidas.

La base de datos distribuida es especialmente deseable porque la administración de una base de datos centralizada está sujeta a problemas como:

- Degradación del rendimiento debido a un creciente número de sitios remotos en grandes distancias.
- Altos costos asociados con mantener y operar sistemas de bases de datos de mainframes.
- Problemas de confiabilidad creados por dependencia de un sitio central y la necesidad de copia de datos.
- Problemas de escalabilidad asociados con los límites físicos impuestos por un sólo factor (potencia, acondicionamiento de temperatura).
- Rigidez organizacional impuesta por la base de datos, que podría no soportar la flexibilidad y agilidad requeridas por organizaciones mundiales modernas.

El dinámico ambiente de negocios y los defectos de las bases de datos centralizadas generaron la demanda de aplicaciones basadas en el acceso a los datos desde diferentes fuentes ubicadas en múltiples lugares. Ese ambiente de múltiples fuentes/ múltiples lugares es manejado mejor por un DBMS distribuido [28].

2.1.2. Promesas de los sistemas distribuidos de bases de datos

Muchas ventajas de los sistemas distribuidos de bases de datos se han citado en la literatura, que van desde razones sociológicas para la descentralización hasta una mejor economía. Todas estas se resumen en cuatro principios que pueden verse como promesas de la tecnología de los sistemas distribuidos de bases de datos: gestión transparente de datos distribuidos y replicados, acceso fiable a datos a través de transacciones distribuidas, desempeño mejorado y facilidad de expansión del sistema [11].

Gestión transparente de datos distribuidos y replicados

La transparencia se refiere a la separación de los aspectos semánticos de alto nivel de un sistema y los aspectos de implementación de bajo nivel. En otras palabras, un sistema transparente "oculta" los detalles de implementación a los usuarios. La ventaja de un DBMS completamente transparente es el alto nivel de soporte que suministra para el desarrollo de aplicaciones complejas. Es obvio que es deseable hacer todos los sistemas (centralizados o distribuidos) completamente transparentes. Por ejemplo, una empresa constructora tiene oficinas en México, Guadalajara y Monterrey, la información de los proyectos se administra en cada uno de estos sitios y se mantiene una base de datos de los empleados, los proyectos y otros datos relacionados. Asumiendo que la base de datos es relacional, es posible almacenar esta información en cuatro relaciones que se muestran en la Figura 2.2. **EMP** guarda información del número, nombre y puesto de los empleados, **PROYECTO** tiene los datos del número, nombre y presupuesto de los proyectos, **SAL** almacena información sobre el salario y **ASG** indica que empleados han sido asignados a que proyectos, con que duración y con que responsabilidad. Si todos estos datos se almacenan en un DBMS centralizado y se quiere encontrar el nombre y el sueldo de los empleados que trabajan en un proyecto por más de doce meses, esto se especifica usando la siguiente consulta SQL:

```
SELECT ENOMBRE, SUELDO
FROM EMP, ASG, SAL
WHERE ASG.DUR>12
AND EMP.ENO=ASG.ENO
AND SAL.PUESTO=EMP.PUESTO
```

Sin embargo, dada la naturaleza distribuida de esta empresa, es preferible, bajo estas circunstancias, ubicar los datos de tal forma que los datos de los empleados en la oficina de México se almacenen en México, aquellos de la oficina de Guadalajara se almacenen en Guadalajara y así sucesivamente. Lo mismo aplica a la información de los proyectos y salarios. Para esto es necesario un proceso donde se fragmenta cada una de las relaciones

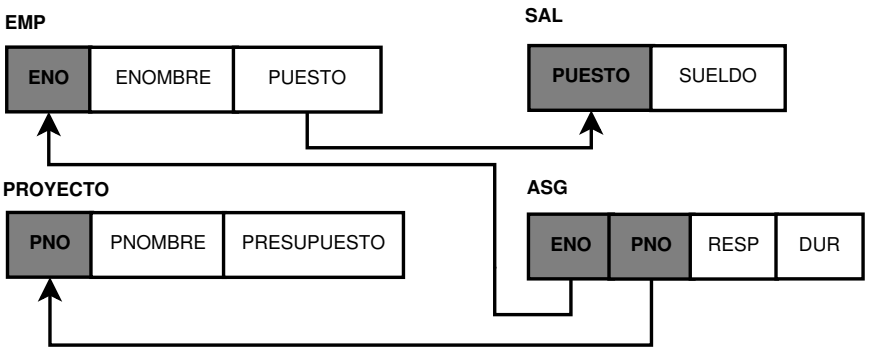


Figura 2.2: Esquema de la base de datos de la empresa constructora

y se almacena cada relación en un sitio diferente. Esto es conocido como fragmentación.

Además podría ser preferible duplicar algunos de estos datos en otros sitios por razones de desempeño y fiabilidad. El resultado es una base de datos distribuida que es fragmentada y replicada (Figura 2.3). El acceso completamente transparente significa que los usuarios todavía pueden realizar la consulta como antes, sin tomar en cuenta la fragmentación, ubicación, o replicación de datos, y dejar que el sistema se preocupe por resolver estos problemas.

Para que un sistema pueda tratar eficientemente este tipo de consultas sobre una base de datos distribuida fragmentada y replicada, es necesario tratar con varios tipos de transparencias.

Independencia de datos La independencia de datos es una forma fundamental de transparencia que se busca en un DBMS. Es además el único tipo importante en el contexto de un DBMS centralizado. Esta se refiere a la inmunidad de las aplicaciones de los usuarios a los cambios en la definición y organización de los datos y viceversa.

La definición de los datos ocurre en dos niveles. En un nivel se especifica la estructura lógica de los datos y en el otro nivel su estructura física. La primera se conoce como la *definición del esquema*, mientras que la última como la *descripción de los datos físicos*. Así que se puede hablar de dos tipos de independencia de datos: independencia lógica

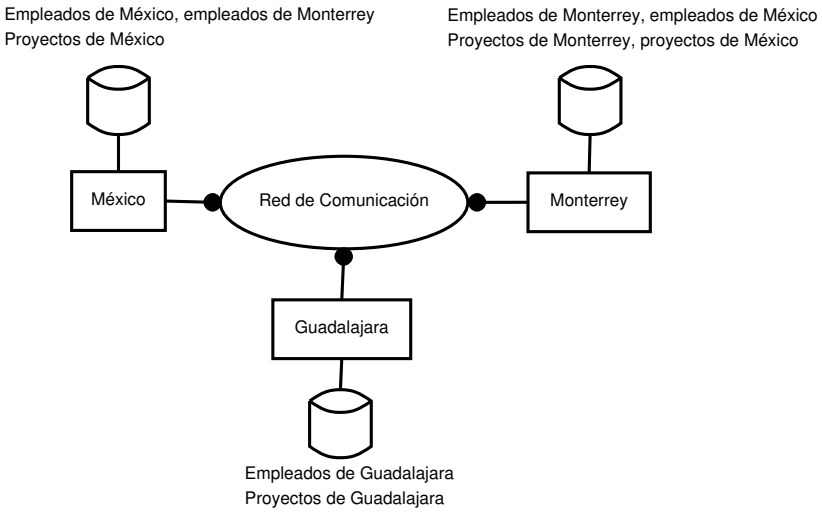


Figura 2.3: Una aplicación distribuida

de datos e independencia física de datos. La *independencia lógica de datos* se refiere a la inmunidad de las aplicaciones de los usuarios a cambios en la estructura lógica (esto es, al esquema) de la base de datos. La *independencia física de los datos*, por otro lado, trata de esconder los detalles de la estructura de almacenamiento a las aplicaciones de los usuarios. Cuando se escribe una aplicación de un usuario, no es necesario preocuparse por los detalles de organización física de los datos. Por lo tanto, las aplicaciones de los usuarios no necesitan modificarse cuando ocurren cambios en la organización de los datos debido a consideraciones de desempeño.

Transparencia de red El usuario no debe preocuparse por los detalles operacionales de la red. Incluso no debe saber que existe una red. No debe haber diferencia entre aplicaciones de bases de datos que se ejecutarían en una base de datos centralizada y aquellas que se ejecutarían en una base de datos distribuida. Este tipo de transparencia se refiere como *transparencia de red* o *transparencia de distribución*.

Se puede considerar a la transparencia de red desde dos puntos de vista: de los servicios provistos o de los datos. Desde la primera perspectiva, es deseable tener medios uniformes

desde los cuales se puede acceder a los servicios. Desde una perspectiva del DBMS, la transparencia de distribución requiere que los usuarios no tengan que especificar donde se localizan los datos.

Algunas veces se identifican dos tipos de transparencia de distribución: transparencia de localización y transparencia de denominación. La transparencia de localización se refiere al hecho de que la orden utilizada para ejecutar una tarea es independiente de la ubicación de los datos y de la ubicación del sistema donde se realizó la orden. La transparencia de denominación implica que, una vez especificado un nombre, puede accederse a los objetos nombrados sin ambigüedad y sin necesidad de especificaciones adicionales.

Transparencia de replicación Por razones de desempeño, fiabilidad y disponibilidad, es usualmente deseable poder distribuir datos en una forma replicada a través de las máquinas de una red. La replicación ayuda al desempeño, ya que se pueden acomodar requerimientos diversos y conflictivos de los usuarios. Por ejemplo, los datos que un usuario accede continuamente pueden colocarse en la máquina local del usuario, así como en la máquina de otro usuario con los mismos requerimientos de acceso. Esto incrementa la localidad de las consultas. Además, si una de las máquinas falla, todavía estará disponible una copia de los datos en otra máquina de la red. Esta es una descripción muy simple de la situación. En realidad, las decisiones sobre replicar o no, y cuántas copias de los objetos de la base de datos se deben tener, depende en un grado considerable de las aplicaciones de los usuarios.

Asumiendo que los datos son replicados, el aspecto de transparencia es si los usuarios deben tener conocimiento de la existencia de copias o si el sistema debería manejar la gestión de las copias y el usuario debería actuar como si hubiera una sola copia de los datos (esto no se refiere a la ubicación de las copias, sólo a su existencia). Desde la perspectiva del usuario, la respuesta es obvia. Es preferible no estar involucrado con el manejo de las copias y no tener que especificar el hecho de que cierta acción puede y/o debería realizarse en múltiples copias. Desde el punto de vista del sistema, la respuesta no es tan simple. Cuando

la responsabilidad de especificar que una acción debe tomarse en múltiples copias se delega al usuario, este hace la gestión de transacciones más simple para el DBMS distribuido. Por otro lado, hacer esto inevitablemente resulta en la pérdida de flexibilidad. El sistema no decide si tener copias o no, y cuántas copias se deben tener, sino las aplicaciones de los usuarios. Cualquier cambio en estas decisiones debido a diversas consideraciones afecta las aplicaciones de los usuarios, y por lo tanto, reduce la independencia de datos significativamente. Por estas razones es deseable que la transparencia de replicación se suministre como una característica estándar del DBMS.

Transparencia de fragmentación La forma final de transparencia que debe tomarse en cuenta en el contexto de un sistema distribuido de bases de datos es la transparencia de fragmentación. Comúnmente es deseable dividir cada relación de la base de datos en fragmentos más pequeños y tratar cada fragmento como un objeto separado de la base de datos (esto es, otra relación). Esto se hace por razones de desempeño, disponibilidad y fiabilidad. Además, la fragmentación puede disminuir los efectos negativos de la replicación. Cada réplica no es la relación completa sino sólo un subconjunto de ella; por lo tanto se requiere menos espacio y es necesario gestionar menos elementos de datos.

Cuando los objetos de la base de datos se fragmentan, es necesario tratar con el problema de manejar consultas de usuarios que se especifican en relaciones completas pero tienen que ejecutarse en subrelaciones. En otras palabras, el problema es encontrar una estrategia de procesamiento de consultas basada en los fragmentos y no en las relaciones, incluso aunque las consultas se especifiquen en las relaciones. Normalmente, esto requiere una traducción de una *consulta global* a varias *consultas sobre los fragmentos.*

Suministro de Transparencia Para proveer acceso fácil y eficiente de los servicios del DBMS a los usuarios nuevos, se desearía tener transparerencia total, involucrando todos los tipos de transparencia. Sin embargo, el nivel de transparencia es inevitablemente un compromiso entre facilidad de uso y la dificultad y costos generales de proveer altos niveles de transparencia. Por ejemplo, Gray argumenta que la transparencia completa hace la

gestión de los datos distribuidos muy difícil y afirma "las aplicaciones que implementan el acceso transparente a datos distribuidos geográficamente tienen: pobre manejabilidad, pobre modularidad y pobre desempeño de mensajes" [29]. Él propone un procedimiento remoto como un mecanismo entre los usuarios y el servidor DBMS por medio del cual los usuarios dirigen sus consultas a un DBMS específico. Esta es la propuesta tomada por los sistemas clientes/servidor.

Pero ¿quién debe ser el encargado de proveer el servicio de transparencia? Es posible identificar tres distintas capas en las que puede suministrarse el servicio de transparencia. Es bastante común tratar éstos como medios mutuamente exclusivos de suministrar el servicio, auque es más apropiado verlos como complementarios.

Se puede dejar a la capa de acceso la responsabilidad de proveer acceso transparente a los recursos de datos. Las características de transparencia pueden construirse en el lenguaje del usuario, quien entonces traduce los servicios requeridos en las operaciones requeridas. En otras palabras, el compilador o el intérprete realizan la tarea y ningún servicio transparente se proporciona al implementador del compilador o intérprete.

La segunda capa en la que puede proporcionarse la transparencia es en el nivel del sistema operativo. Los sistemas operativos del estado del arte proveen algún nivel de transparencia a los usuarios del sistema. Por ejemplo, los controladores de los dispositivos en los sistemas operativos manejan los detalles de lograr que cada pieza del equipo periférico realice sus funciones. El usuario de la computadora típico o incluso un programador de aplicaciones, normalmente no implementa controladores de dispositivos para interactuar con equipo periférico individual; esa operación es transparente para el usuario.

Proveer acceso transparente a los recursos en el nivel del sistema operativo puede extenderse al ambiente distribuido, donde la gestión de los recursos de la red es llevada a cabo por el sistema operativo distribuido o el middleware si es que el DBMS distribuido se implementa sobre uno. Hay dos problemas potenciales con esta propuesta. El primero es que no todos los sistemas operativos distribuidos proveen un nivel razonable de transparencia en cuanto a la gestión de la red. El segundo problema es que algunas aplicaciones no

desean que se les oculten los detalles de la distribución y necesitan accederlos para aspectos específicos de afinación de desempeño.

La tercera capa en la que puede soportarse la transparencia es en el DBMS. La transparencia y el soporte para las funciones de la base de datos suministrado a los diseñadores del DBMS por un sistema operativo subyacente es generalmente mínimo y limitado a operaciones fundamentales para ejecutar ciertas tareas. Es responsabilidad del DBMS hacer todas las traducciones necesarias desde el sistema operativo hasta la interfaz de usuario de alto nivel. Este modo de operación es el método más común actualmente. Sin embargo, hay varios problemas asociados con el hecho de dejar la tarea de proveer la transparencia completa al DBMS. Éstos tienen que ver con la interacción del sistema operativo con el DBMS distribuido.

Una jerarquía de estas transparencias se presenta en la Figura 2.4. La capa de transparencia de acceso se refiere a que los usuarios tienen acceso de alto nivel a los datos (por ejemplo, lenguajes de cuarta generación, interfaces gráficas de usuario, acceso a través de lenguaje natural).

Fiabilidad por medio de transacciones distribuidas

Los DBMSs distribuidos permiten mejorar la fiabilidad debido a que tienen componentes replicados y, por lo tanto, eliminan los puntos individuales de fallo. El fallo de un sitio, o el fallo de un enlace de comunicación que hace uno o más sitios inalcanzables, no es suficiente para derribar el sistema completo. En el caso de una base de datos distribuida, esto significa que algunos de los datos puede ser inalcanzables, pero con el cuidado adecuado, los usarios todavía pueden acceder a otras partes de la base de datos distribuida. El "cuidado adecuado" viene en la forma de soporte para transacciones distribuidas y protocolos de aplicación.

Una transacción es una unidad básica de procesamiento consistente y fiable, que incluye una secuencia de operaciones de la base de datos ejecutadas como una acción atómica. Cada transacción debe dejar a la base de datos en un estado consistente, incluso cuando

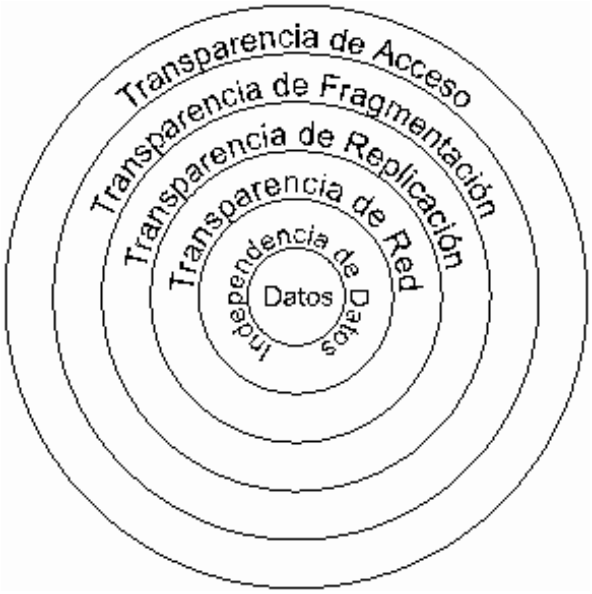


Figura 2.4: Capas de transparencia

varias transacciones se ejecuten concurrentemente (algunas veces llamada transparencia de concurrencia) e incluso cuando ocurran fallos (también llamada atomicidad de fallo). Por lo tanto, un DBMS que provee soporte completo de transacciones garantiza que la ejecución concurrente de las transacciones de los usuarios no violará la consistencia de la base de datos al enfrentarse a fallos del sistema siempre y cuando cada transacción sea correcta, esto es, cumpla con las reglas de integridad especificadas en la base de datos.

Por ejemplo, en la empresa de construcción anteriormente descrita, asumiendo que hay una aplicación que incrementa los salarios de los empleados en un 10 %, es deseable encapsular la consulta (o el código del programa) que realiza esta tarea dentro de los límites de una transacción. Por ejemplo, si un fallo del sistema ocurre a la mitad de la ejecución de este programa, sería deseable que el DBMS pueda determinar, después de una recuperación, dónde se quedó y pueda continuar con su operación (o iniciar todo otra vez). Esto se refiere a atomicidad de fallo. Alternativamente, si algún otro usuario ejecuta una consulta que calcule el promedio de los salarios de los empleados en esa compañía mientras la primera acción se está llevando a cabo, el resultado calculado será erróneo. Por lo tanto, es deseable

que el sistema pueda sincronizar la ejecución concurrente de estos dos programas. Para encapsular una consulta (o un código de programa) en los límites de una transacción, es suficiente declarar el inicio de la transacción y su final.

```
BEGIN TRANSACTION actualiza_salario
    UPDATE SAL SET SUELDO=SUELDO*1.1;
END TRANSACTION
```

Las transacciones distribuidas se ejecutan en varios sitios, cada uno de los cuales forma una base de datos local. La transacción anterior, por ejemplo, se ejecutará en México, Guadalajara y Monterrey, ya que los datos se encuentran distribuidos en todos estos sitios. Con soporte completo para transacciones distribuidas, las aplicaciones de los usuarios pueden acceder a una imagen lógica simple de la base de datos y confiar en el DBMS distribuido para asegurar que sus solicitudes se ejecutarán correctamente sin importar lo que pase en el sistema. "Correctamente" significa que las aplicaciones del usuario no necesitan preocuparse por coordinar su acceso a bases de datos locales individuales ni preocuparse acerca de que un sitio esté disponible o que el enlace de comunicación falle durante la ejecución de sus transacciones.

Desempeño mejorado

Un DBMS distribuido mejora el desempeño de las consultas debido a dos factores. Primero, porque fragmenta la base de datos conceptual, permitiendo que los datos se almacenen los más cerca posible del sitio donde más se requieren (también llamada localidad de datos). Esto tiene dos ventajas potenciales:

1. Debido a que cada sitio maneja sólo una porción de la base de datos, la competencia por los servicios de CPU y E/S no son tan severos como en las bases de datos centralizadas.

2. La localidad reduce retrasos de acceso remoto que se involucran en redes de área extendida (por ejemplo, el retraso mínimo de la propagación de mensajes de ida y

vuelta en los sistemas basados en satélites es de aproximadamente un segundo).

La mayoría de los DBMSs distribuidos se estructuran para ganar el beneficio máximo de la localidad. Los beneficios completos de la competencia y costos de comunicación reducidos pueden obtenerse sólo por medio de una fragmentación y distribución adecuada de la base de datos.

Este punto se relaciona con los costos generales del cómputo distribuido si los datos tienen que residir en sitios remotos y se deben acceder por medio de comunicación remota. El argumento es que es mejor, en estas circunstancias, distribuir la funcionalidad de la gestión de los datos al lugar donde se ubican los datos en vez de mover grandes cantidades de datos. Esto ha sido un tema de discusión recientemente. Algunos afirman que con el uso extensivo de redes de alta velocidad y alta capacidad, las funciones de gestión y distribución de datos ya no tienen sentido y que puede ser mucho más simple almacenar datos en un sitio central y accederlo (realizando descargas) por medio de redes de alta velocidad. Este argumento, aunque es interesante, ignora el objetivo de las bases de datos distribuidas. Primero que nada, en la mayoría de las aplicaciones de hoy, los datos están distribuidos; lo que puede abrirse a debate es cómo y dónde procesarlos. Segundo, y más importante, es que este argumento no distingue entre banda ancha (la capacidad de enlaces de computadoras) y latencia (cuánto tiempo toma transmitir los datos). La latencia es inherente en los ambientes distribuidos y hay límites físicos de la rapidez con la que se pueden enviar datos sobre redes de computadoras. Como se indicó anteriormente, por ejemplo, los enlaces de satélites toman casi medio segundo en transmitir datos entre dos estaciones terrestres. Esto se debe a la distancia existente entre los satélites y la tierra, por lo tanto no hay nada que se pueda hacer para mejorar el desempeño. Para algunas aplicaciones, esto podría significar un retraso inaceptable.

Un tercer punto es que el paralelismo inherente de sistemas distribuidos puede explotarse por medio de paralelismo entre consultas y dentro de las consultas. El paralelismo entre consultas resulta de la habilidad para ejecutar múltiples consultas al mismo tiempo, mientras que el paralelismo dentro de las consultas se logra dividiendo una consulta simple

en varias subconsultas, cada una de las cuales se ejecuta en un sitio diferente accediendo a una parte distinta de la base de datos distribuida.

Si el acceso de los usuarios a la base de datos distribuida consiste sólo de consultas (esto es, acceso de sólo lectura), entonces la provisión de paralelismo entre consultas y dentro de las consultas implicaría que la base de datos debería replicarse lo más posible. Sin embargo, ya que la mayoría de los accesos a la base de datos no son de sólo lectura, la mezcla de operaciones de lectura y actualización requiere la implementación de control de concurrencia distribuida.

Facilidad de expansión del sistema.

En un ambiente distribuido, es mucho más fácil incrementar el tamaño de la base de datos. Revisiones importantes del sistema son rara vez necesarias; la expansión puede manejarse agregando poder de procesamiento y de almacenamiento a la red. Obviamente, puede no ser posible obtener un incremento lineal en "poder", ya que esto también depende de los costos generales de la distribución. Sin embargo, todavía son posibles mejoras importantes.

Un aspecto de facilidad de expansión del sistema es la economía. Normalmente cuesta mucho menos formar un sistema de computadoras "más pequeñas" con el poder equivalente de una sola máquina grande. En tiempos anteriores, se creía que se podría comprar una computadora que cuaduplicara su poder gastando al menos el doble. Esto se conoce como la ley de Grosh. Con el surgimiento de microcomputadoras y estaciones de trabajo y sus características de precio y rendimiento, esta ley se considera inválida [11].

2.1.3. Complicaciones introducidas por la distribución

Los problemas encontrados en los sistemas de bases de datos toman una complejidad adicional en un ambiente distribuido, incluso aunque los principios fundamentales son los mismos. Además, esta complejidad adicional origina nuevos problemas influenciados principalmente por tres factores.

Primero, los datos pueden replicarse en un ambiente distribuido. Una base de datos

distribuida puede diseñarse de tal forma que la base de datos completa, o partes de ella, residan en sitios diferentes de una red de computadoras. No es esencial que cada sitio de la red contenga la base de datos; sólo es esencial que haya más de un sitio donde resida la base de datos. La duplicación posible de elementos de datos se debe principalmente a las consideraciones de eficiencia y fiabilidad. Consecuentemente, el sistema distribuido de bases de datos es reponsable de (1) elegir qué copias almacenadas de los datos solicitados deben accederse en caso de recuperaciones, y (2) asegurarse de que el efecto de una actualización se refleje en cada una de las copias que contengan al elemento de datos actualizado.

Segundo, si algunos sitios fallan (por ejemplo, ya sea por el mal funcionamiento de hardware o software) o si algunos enlaces de comunicación fallan (haciendo inalcanzables algunos de los sitios) mientras se ejecuta una actualización, el sistema debe asegurarse que los efectos se reflejarán en los datos que residen en los sitios inalcanzables tan pronto como el sistema pueda recuperarse del fallo.

El tercer punto es que debido a que cada sitio no puede tener información instantánea de las acciones actualmente llevadas a cabo en los otros sitios, la sincronización de transacciones en múltiples sitios es considerablemente más difícil que en un sistema centralizado.

Estas dificultades apuntan a varios problemas potenciales en DBMSs distribuidos. Estos son la inherente complejidad de construir aplicaciones distribuidas: diseño de la base de datos distribuida, gestión del directorio distribuido, procesamiento distribuido de consultas, control de concurrencia distribuida, fiabilidad del DBMS distribuido, replicación [11].

Diseño de la base de datos distribuida

El problema a resolver es cómo ubicar a través de los sitios a la base de datos y a las consultas que se ejecutan en ella. Hay dos alternativas básicas para colocar los datos: *fragmentados* (o *no replicados*) y *replicados*. En el esquema fragmentado la base de datos se divide en varias particiones disjuntas cada una de las cuales se coloca en un sitio diferente. Los diseños replicados pueden ser *completamente replicados* (también llamados *completamente duplicados*) donde la base de datos completa se almacena en cada sitio, o *parcial-*

mente replicados (o *parcialmente duplicados*) donde cada partición de la base de datos se almacena en más de un sitio, pero no en todos los sitios. Los dos problemas de diseño fundamentales son la *fragmentación*, esto es, la separación de la base de datos en particiones llamadas *fragmentos*, y *distribución*, la ubicación óptima de los fragmentos.

La investigación en esta área involucra principalmente programación matemática con el fin de minimizar el costo combinado de almacenamiento de la base de datos, procesamiento de transacciones, y comunicación de mensajes entre sitios. El problema general es NP-hard. Por lo tanto, las soluciones propuestas se basan en heurísticas.

Gestión del directorio distribuido

Un directorio contiene información (tal como descripciones y ubicaciones) acerca de los elementos de datos de la base de datos. Los problemas relacionados con la gestión del directorio son similares en naturaleza al problema de la ubicación de los datos descrito anteriormente. Un directorio puede ser global del sistema distribuido de bases de datos o local de cada sitio; puede centralizarse en un sitio o distribuirse sobre varios sitios; puede haber una sola copia o múltiples copias.

Procesamiento distribuido de consultas

El procesamiento de consultas trata con el diseño de algoritmos que analizan consultas y las convierten en un conjunto de operaciones de manipulación de datos. El problema es cómo decidir una estrategia para ejecutar cada consulta a través de la red de la forma más eficiente posible. Los factores a considerar son la distribución de datos, costos de comunicación y falta de suficiente información disponible localmente. El objetivo es determinar la mejor forma de utilizar el paralelismo inherente para mejorar el desempeño de la ejecución de la transacción, tomando en cuenta las restricciones anteriores. El problema es NP-hard en naturaleza, y las propuestas son usualmente heurísticas.

Control de concurrencia distribuida

El control de concurrencia involucra la sincronización de los accesos a la base de datos distribuida, de tal forma que se mantenga la integridad de la base de datos. Este es, sin lugar a dudas, uno de los problemas más estudiado en el campo de los sistemas distribuidos de bases de datos. El problema de control de concurrencia en un contexto distribuido es un poco diferente que en la estructura centralizada. Uno no sólo se tiene que preocupar acerca de la integridad de una sola base de datos, sino también acerca de la consistencia de múltiples copias de la base de datos. La condición que requiere que todos los valores de múltiples copias de cada elemento de datos converja al mismo valor es llamada *consistencia mutua*.

Las soluciones alternativas se dividen en dos clases generales: *pesimistas* y *optimistas*. Las primeras sincronizan la ejecución de las consultas de los usuarios antes de que inicie la ejecución, las segundas primero ejecutan las solicitudes y después verifican que la ejecución mantenga la consistencia de la base de datos. Dos primitivas fundamentales que pueden usarse con las dos propuestas son *bloqueos*, que se basan en la exclusión mutua de los accesos a los elementos de datos, y *sellado de tiempo*, donde las ejecuciones de las transacciones se ordenan con base en estampas de tiempo. Hay variaciones de estos esquemas, así como algoritmos híbridos que intentan combinar los dos mecanismos básicos.

Fiabilidad del DBMS distribuido

Una de las ventajas potenciales de sistemas distribuidos es mejorar la fiabilidad y disponibilidad. Sin embargo, esta ventaja no ocurre automáticamente. Es importante que se suministren mecanismos para asegurar la consistencia de la base de datos, así como detectar fallos y recuperarse de ellos. La implicación del sistema distribuido de bases de datos es que cuando un fallo ocurre y varios sitios se vuelven inoperables o inaccesibles, las bases de datos en los sitios operacionales permanezcan consistentes y actualizadas. Además, cuando el sistema de cómputo o la red se recuperan del fallo, el sistema distribuido de bases

de datos debería poder recuperarse y actualizar las bases de datos en los sitios que fallaron. Esto puede ser especialmente difícil en el caso de la partición de la red, donde los sitios se dividen en dos o más grupos sin comunicación entre ellos.

Replicación

Si la base de datos distribuida es (parcialmente o totalmente) replicada, es necesario implementar protocolos que aseguren la consistencia de las réplicas, esto es, que las copias de los mismos elementos de datos tengan los mismos datos. Estos protocolos pueden ser *entusiastas,* ya que obligan a que las actualizaciones se apliquen a todas las réplicas antes de que se complete una transacción, o pueden ser *perezosos* de tal forma que la transacción actualiza una copia (llamada *maestra*) de la cual se propagan las actualizaciones a las otras después de que se completa la transacción.

Relación entre problemas

Naturalmente, estos problemas no están aislados uno del otro. Cada problema es afectado por las soluciones encontradas para los otros, y a la vez esto afecta el conjunto de soluciones factibles para ellos. La relación entre los componentes se muestra en la Figura 2.5. El diseño de las bases de datos distribuidas afecta muchas áreas. Este afecta la gestión del directorio, porque la definición de fragmentos y su ubicación determina el contenido del directorio (o directorios), así como las estrategias que pueden emplearse para manejarlas. La misma información (esto es, la estructura y ubicación de los fragmentos) se usa por el procesador de consultas para determinar la estrategia de evaluación de consultas. Por otro lado, los patrones de acceso y uso que se determinan por el procesador de consultas se usan como entrada en los algoritmos de fragmentación y distribución de datos. De igual forma, la ubicación y el contenido del directorio influye en el procesamiento de consultas.

La replicación de fragmentos cuando ellos están distribuidos afecta las estrategias de control de concurrencia que podrían emplearse. Algunos algoritmos de control de concurrencia no pueden usarse fácilmente en bases de datos replicadas. De igual forma, los patrones

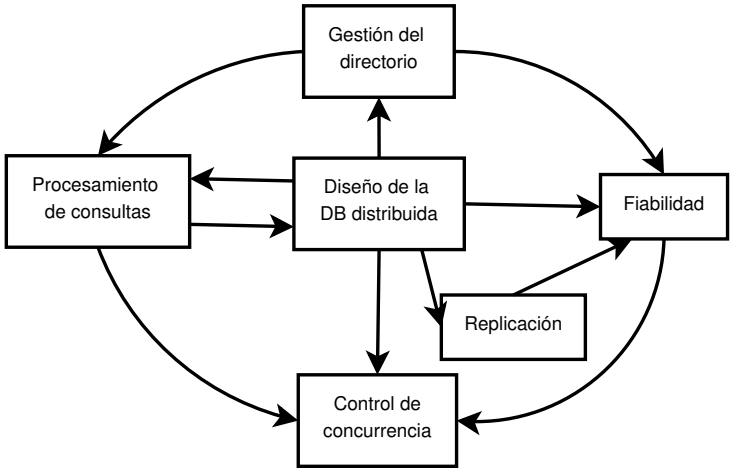


Figura 2.5: Relación entre los problemas de investigación.

de acceso y uso de la base de datos influyen en los algoritmos de control de concurrencia. Si en el ambiente ocurren muchas actualizaciones, las precauciones necesarias son bastante diferentes de aquellas que deben tomarse en ambientes de consultas de sólo lectura.

Los mecanismos de fiabilidad involucran técnicas de recuperación local y protocolos de fiabilidad distribuida. En ese sentido, ellas influyen en la opción de las técnicas de control de concurrencia que se construyen sobre ellas. La técnicas para proveer fiabilidad también usan la información de la ubicación de los datos debido a que la existencia de copias duplicadas de datos sirven como una garantía para mantener la operación fiable de la base de datos distribuida.

Finalmente, surge la necesidad de protocolos de aplicación si la distribución de datos involucra réplicas. Hay una fuerte relación entre protocolos de replicación y técnicas de control de concurrencia, ya que ambas tratan con la consistencia de los datos, pero desde diferentes perspectivas. Por lo tanto, los protocolos de replicación influyen en las técnicas de fiabilidad distribuida tales como protocolos de compromiso. De hecho, algunas veces se sugiere que los protocolos de replicación puedan usarse, en vez de implementar protocolos de compromiso distribuidos.

Problemas adicionales

Los problemas de diseño anteriores cubren lo que puede llamarse sistemas distribuidos de bases de datos "tradicionales". El ambiente ha cambiado significativamente debido a que estos temas empezaron a investigarse planteando oportunidades y retos adicionales.

Uno de los desarrollos importantes ha sido el movimiento hacia una federación "más suelta" entre fuentes de datos, que pueden además ser heterogéneos. Esto originó el desarrollo de sistemas de bases de datos federadas o sistemas de integración de datos que requieren una nueva investigación de algunas de las técnicas de bases de datos fundamentales. Estos sistemas constituyen una parte importante del ambiente distribuido de hoy.

El crecimiento de Internet como una plataforma de red fundamental ha originado preguntas importantes sobre las suposiciones de los sistemas distribuidos de bases de datos subyacentes. Dos aspectos son de particular interés. Uno es el resurgimiento del cómputo punto a punto, y la otra es el desarrollo y crecimiento del World Wide Web. Ambos tienen como objetivo mejorar la compartición de datos, pero toman diferentes propuestas y representan diferentes retos en la gestión de los datos.

Finalmente, hay una fuerte relación entre bases de datos distribuidas y paralelas. Aunque las primeras asumen que cada sitio es una computadora lógica simple, una gran parte de las instalaciones son, de hecho, clusters paralelos. Por lo tanto, aunque la mayoría de la literatura se enfoca en los problemas que surgen en la gestión de los datos distribuidos a través de varios sitios, problemas de gestión de datos interesantes existen en un sólo sitio lógico que puede ser un sistema paralelo.

2.1.4. Tecnología distribuida versus tecnología paralela

Hay dos principales tipos de arquitecturas de sistemas multiprocesador:

- **Arquitectura de memoria compartida (fuertemente acoplada):** Múltiples procesadores comparten almacenamiento secundario (disco) y memoria principal.
- **Arquitectura de disco compartido (débilmente acoplada):** Múltiples proce-

sadores comparten almacenamiento secundario (disco) pero cada uno tiene su propia memoria principal.

Estas arquitecturas permiten a los procesadores comunicarse sin el costo de intercambiar mensajes sobre una red. Los sistemas gestores de bases de datos desarrollados usando los tipos de arquitecturas anteriores se llaman DBMSs paralelos en vez de DBMSs distribuidos, ya que utilizan tecnología de procesadores paralelos. Otro tipo de arquitectura de múltiples procesadores es la llamada arquitectura "nada compartido". En esta arquitectura, cada procesador tiene su propia memoria primaria y secundaria (disco), no existe memoria común, y los procesadores se comunican a través de redes de alta velocidad. Aunque la arquitectura de nada compartido se asemeja al ambiente de una base de datos distribuida, las principales diferencias existen en el modo de operación. En los sistemas multiprocesador de nada compartido, hay simetría y homogeneidad de nodos; mientras que en el ambiente de las bases de datos distribuidas es muy común la heterogeneidad de hardware y de sistemas operativos en cada nodo. La arquitectura de nada compartido también se considera como un ambiente para bases de datos paralelas [26].

2.2. Diseño de bases de datos distribuidas

El diseño de un sistema de cómputo distribuido involucra tomar decisiones sobre la colocación de los datos y los programas a través de los sitios de una red de computadoras. En el caso de DBMSs distribuidos, la distribución de aplicaciones involucra dos cosas: la distribución del software del DBMS distribuido y la distribución de los programas de aplicación que se ejecutan en él. La organización de los sistemas distribuidos puede investigarse a través de tres dimensiones ortogonales [30] (Figura 2.6).

1. Nivel de compartición
2. Comportamiento de patrones de acceso
3. Nivel de conocimiento sobre el comportamiento de los patrones de acceso

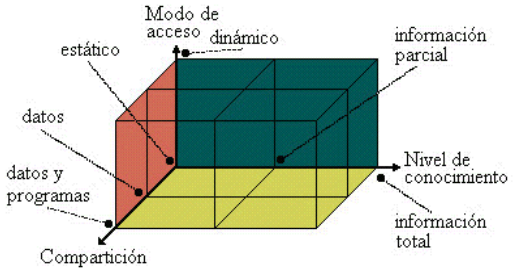


Figura 2.6: Estructura de distribución

En términos del nivel de compartición hay tres posibilidades: primero *no hay compartición*: cada aplicación y sus datos se ejecutan en un solo sitio, y no hay comunicación con ningún otro programa o acceso a ningún archivo de datos en ningún otro sitio. Esto caracteriza los inicios de las redes y probablemente no es muy común hoy en día. En el nivel de *compartición de datos*; todos los programas se replican en todos los sitios, pero los archivos de datos no. De acuerdo a esto, las solicitudes de los usuarios se manejan en el sitio donde se originan y los archivos de datos necesarios se mueven a través de la red. Finalmente, en el *nivel de compartición de datos mas programas*, se comparten tanto los datos como los programas, esto significa que un programa en un sitio dado puede solicitar un servicio de otro programa en un segundo sitio, el cual a la vez puede tener acceso a datos ubicados en un tercer sitio.

Levin y Morgan [30] hicieron una distinción entre compartición de datos y compartición de datos mas programas para ilustrar las diferencias entre sistemas de cómputo distribuidos homogéneos y heterogéneos. Ellos indican, correctamente, que en un ambiente heterogéneo es normalmente muy difícil, y algunas veces hasta imposible, ejecutar un programa en un hardware diferente bajo un sistema operativo diferente. Sin embargo, podría ser relativamente fácil mover datos.

A lo largo de la segunda dimensión del comportamiento de los patrones de acceso, es posible identificar dos alternativas. Los patrones de acceso de las consultas de los usuarios pueden ser *estáticos*, de tal forma que ellos no cambian a través del tiempo, o *dinámicos*.

Obviamente es considerablemente más fácil programar y manejar los ambientes estáticos de lo que sería manejar el caso de los sistemas distribuidos dinámicos. Desafortunadamente, es difícil encontrar muchas aplicaciones distribuidas reales que podrían clasificarse como estáticas. La pregunta importante, entonces, no es si un sistema es estático o dinámico, sino que tan dinámico es. A través de esta dimensión se establece la relación entre el diseño de la base de datos distribuida y el procesamiento de consultas.

La tercera dimensión de clasificación es el nivel de conocimiento acerca de los patrones de acceso. Una posibilidad es que los diseñadores no tengan ninguna información sobre cómo los usuarios accederán a la base de datos. Esta es una posibilidad teórica, pero es muy difícil, si no es que imposible, diseñar un DBMS distribuido que pueda manejar esta situación. Las alternativas más prácticas son que los diseñadores tengan *información completa*, donde los patrones de acceso pueden predecirse razonablemente, y no desviarse significativamente de estas predicciones, o *información parcial*, donde hay desviaciones de las predicciones.

El problema del diseño de bases de datos distribuidas debe considerarse dentro de esta estructura general. En todos los casos discutidos, excepto en la alternativa de no compartición, se presentan nuevos problemas que no ocurren en bases de datos centralizadas.

Se han identificado dos estrategias principales para diseñar bases de datos distribuidas, la propuesta *descendente* y *ascendente* [31]. La propuestas descendente es más adecuada para DBMSs distribuidos homogéneos fuertemente integrados, mientras que el diseño ascendente es más adecuado para bases de datos múltiples.

2.2.1. Estrategia de diseño descendente

Una estructura para el proceso de diseño descendente se muestra en la Figura 2.7. La actividad inicia con un análisis de requerimientos que define el ambiente del sistema y "obtiene tanto los datos como las necesidades de procesamiento de todos los usuarios potenciales de la base de datos" [32]. El estudio de requerimientos también permite determinar qué objetivos de un DBMS distribuido debe cumplir el sistema final. Estos objetivos

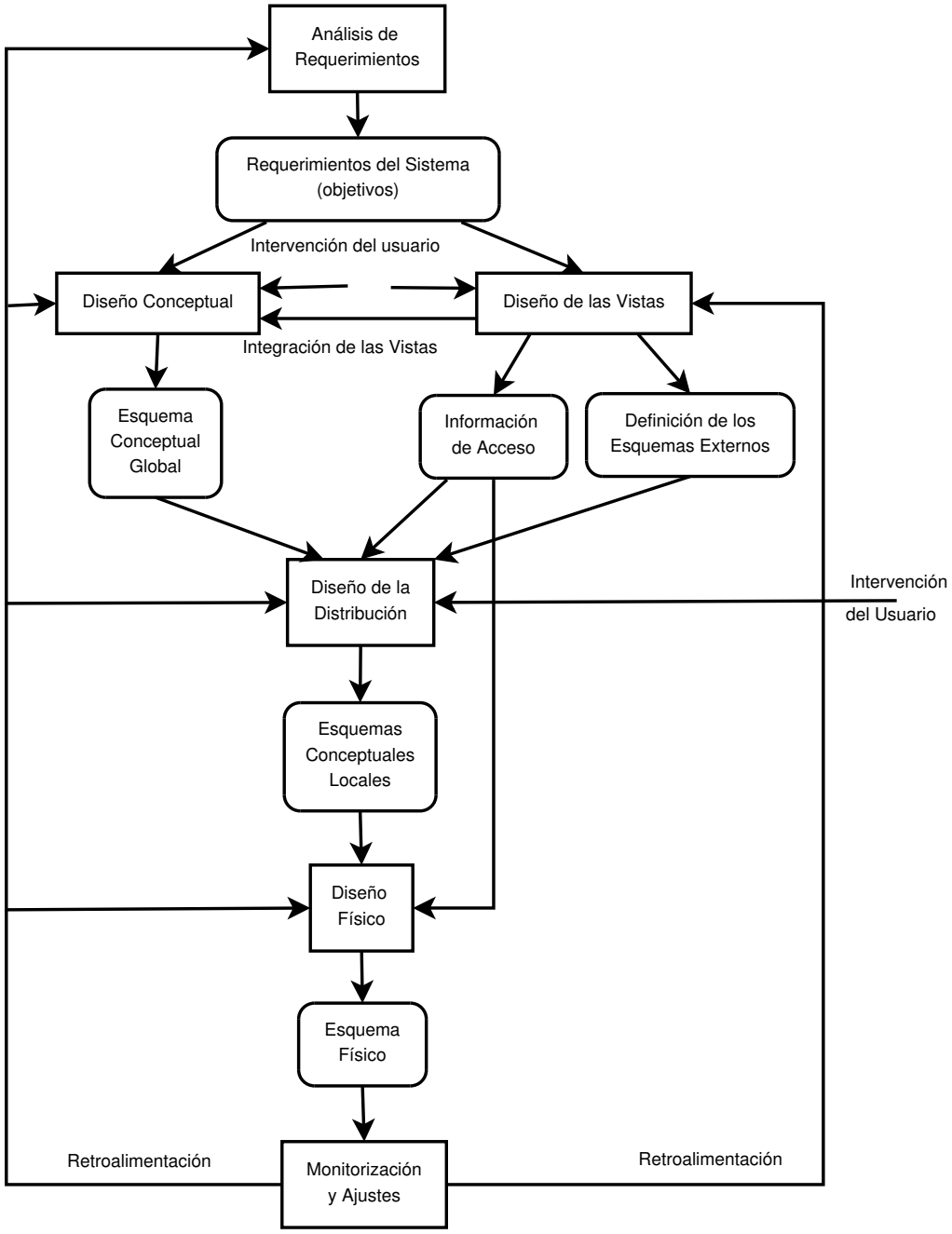


Figura 2.7: Proceso de diseño descendente

se definen con respecto a desempeño, fiabilidad y disponibilidad, economía y facilidad de expansíon (flexibilidad).

El documento de requerimientos es la entrada a dos actividades paralelas: el diseño de las vistas y el diseño conceptual. La actividad del *diseño de las vistas* trata con la definición de interfaces para los usuarios finales. El *diseño conceptual*, por otro lado, es el proceso por medio del cual la empresa se examina para determinar tipos de entidades y relaciones entre estas entidades.

Hay una relación entre el diseño conceptual y el diseño de las vistas. El diseño conceptual puede interpretarse como la *integración de las vistas* del usuario. Incluso, aunque esta actividad de integración de vistas es muy importante, el modelo conceptual no debe soportar sólo las aplicaciones existentes, sino también las aplicaciones futuras. La integración de las vistas debe usarse para asegurar que los requerimientos de relaciones y entidades para todas las vistas se cubren en el esquema conceptual.

En las actividades de diseño conceptual y diseño de las vistas el usuario necesita especificar las entidades de datos y debe determinar las aplicaciones que se ejecutarán en la base de datos, así como la información estadística acerca de estas aplicaciones. Dicha información incluye la especificación de la frecuencia de las aplicaciones de los usuarios, el volumen de diversa información y demás. Del paso de diseño conceptual viene la definición del esquema conceptual global. Hasta este punto, el proceso es idéntico al diseño de una base de datos centralizada.

El Esquema Conceptual Global (GCS) y la información de los patrones de acceso recolectada como resultado del diseño de las vistas son entradas al paso del *diseño de la distribución*. El objetivo en esta fase, es diseñar los esquemas conceptuales locales (LCSs) por medio de la distribución de las entidades sobre los sitios del sistema distribuido. También es posible tratar cada entidad como una unidad de distribución. En el modelo relacional las entidades corresponden a relaciones.

En vez de distribuir relaciones, es bastante común dividirlas en subrelaciones llamadas fragmentos, que posteriormente se distribuyen. Entonces, la actividad del diseño de la

distribución consiste en dos pasos: fragmentación y asignación. La razón para separar el diseño de la distribución en dos pasos es para tratar mejor con la complejidad del problema.

El último paso es el diseño físico, que mapea los LCSs a los dispositivos de almacenamiento físico disponibles en los sitios correspondientes. Las entradas a este proceso son los LCSs y la información de los patrones de acceso a los fragmentos.

Es bien conocido que las actividades de diseño y desarrollo de cualquier tipo son procesos en curso que requieren constante monitorización, ajustes y afinaciones periódicas. Por lo tanto, se incluye la observación y la monitorización como una actividad principal en este proceso. No sólo se monitoriza el comportamiento de la implementación de la base de datos sino también lo adecuado de las vistas de los usuarios. El resultado es alguna forma de retroalimentación, que puede resultar en el regreso a uno de los pasos anteriores en el diseño.

2.2.2. Estrategia de diseño ascendente

Ceri y Pelagati [33] y Özsu y Valduriez [11] afirmaron que el diseño descendente es adecuado para sistemas que inician desde cero. Pero cuando la base de datos distribuida se desarrolla como la reunión de bases de datos existentes, no es fácil seguir la propuesta descendente. La propuesta ascendente que inicia con esquemas conceptuales locales individuales, es más adecuada para este ambiente. Ceri y Pelagati explicaron que la propuesta ascendente se basa en la integración de los esquemas existentes en un solo esquema global. La integración es el proceso de fusionar las definiciones de datos comunes y la resolución de conflictos entre diferentes representaciones que se dan a los mismos datos. El GCS es el producto de estos procesos [11]. Ceri y Pelagati concluyeron que hay tres requerimientos para un diseño ascendente:

1. La selección de un modelo de bases de datos común para describir el esquema global de la base de datos.

2. La traducción de cada esquema local en el modelo de datos común.

3. La integración del esquema local en el esquema global.

2.2.3. Problemas del diseño de la distribución

Anteriormente se indicó que las relaciones en un esquema de base de datos se descomponen usualmente en fragmentos más pequeños, pero no se ofreció ninguna justificación o detalles de este proceso. A continuación se presentarán estos detalles. Las preguntas que se responderán son:

1. ¿Por qué se debe fragmentar?
2. ¿Cómo se debe fragmentar?
3. ¿Hasta qué grado se debe fragmentar?
4. ¿Hay alguna forma de probar la correcta fragmentación?
5. ¿Cómo se deben asignar los fragmentos?
6. ¿Cuál es la información necesaria para realizar la fragmentación y la asignación?

Razones para fragmentar

Desde el punto de vista de distribución de datos, no hay razón para fragmentar los datos. Despúes de todo, en sistemas de archivos distribuidos se realiza una distribución de los archivos completos. Los primeros trabajos sobre diseño de la distribución trataban con la asignación de archivos a nodos de una red de computadoras.

Con respecto a la fragmentación, el problema importante es la unidad de distribución apropiada. Una relación no es una adecuada unidad de distribución por varias razones. Primero, las vistas de las aplicaciones usualmente son subconjuntos de relaciones. Por lo tanto, la localidad de los accesos de las aplicaciones no se define sobre relaciones completas sino sobre sus subconjuntos. Por lo tanto, es natural considerar subconjuntos de relaciones como unidades de distribución.

Segundo, si las aplicaciones que tienen vistas definidas en una relación dada residen en diferentes sitios, pueden seguirse dos alternativas, con la relación completa como unidad de distribución. La relación no se replica y se almacena en un sólo sitio, o se replica en todos o algunos de los sitios donde residen las aplicaciones. La primera resulta en un alto volumen innecesario de accesos a datos remotos. La segunda, por otro lado, tiene replicación innecesaria, esto causa problemas en la ejecución de actualizaciones y puede no ser deseable si el almacenamiento es limitado.

Finalmente, la descomposición de una relación en fragmentos, cada uno tratado como una unidad, permite que varias transacciones se ejecuten concurrentemente. Además, la fragmentación de relaciones normalmente resulta en la ejecución paralela de una consulta dividiéndola en un conjunto de subconsultas que operan sobre los fragmentos. Por lo tanto, la fragmentación normalmente incrementa el nivel de concurrencia y, por lo tanto, el rendimiento del sistema.

Con la fragmentación surgen dificultades también. Si las consultas tienen requerimientos conflictivos que impiden la descomposición de la relación en fragmentos mutuamente exclusivos, aquellas aplicaciones que acceden a los datos ubicados en más de un fragmento pueden sufrir degradación de desempeño. Podría ser necesario, por ejemplo, recuperar datos de dos fragmentos y entonces hacer su reunión, la cual es muy costosa. Un problema fundamental de la fragmentación es minimizar las reuniones distribuidas.

El segundo problema se relaciona con el control semántico de datos, específicamente con verificar la integridad. Como un resultado de la fragmentación, los atributos que participan en una dependencia pueden descomponerse en diferentes fragmentos que podrían asignarse a sitios diferentes [11].

Alternativas de fragmentación

Las instancias de las relaciones son esencialmente tablas, así que el problema es encontrar formas de dividir la tabla en relaciones más pequeñas. Hay tres alternativas para esto: fragmentación horizontal, fragmentación vertical y fragmentación mixta o híbrida [28], las

cuales se explicarán con más detalle en el siguiente Capítulo.

Grado de fragmentación

El grado en el que debe fragmentarse una base de datos es una decisión importante que afecta el desempeño de la ejecución de las consultas. El grado de fragmentación va de un extremo, que es, no fragmentar, al otro extremo, fragmentar al nivel de tuplas individuales (en el caso de la fragmentación horizontal) o al nivel de atributos individuales (en el caso de la fragmentación vertical).

Ya se discutieron los efectos adversos de las unidades de fragmentación muy grandes o muy pequeñas. Es necesario encontrar un nivel adecuado de fragmentación que sea un compromiso entre estos dos extremos. Dicho nivel sólo se puede definir con base en las consultas que se ejecutarán en la base de datos. El problema es, ¿cómo? En general, las consultas tienen que describirse con respecto a varios parámetros. De acuerdo a los valores de estos parámetros, se pueden identificar los fragmentos individuales.

Reglas de corrección de la fragmentación

Se deben cumplir las siguientes tres reglas durante la fragmentación, las cuales en conjunto aseguran que la base de datos no experimente cambios semánticos durante la fragmentación.

1. *Completitud.* Si una instancia de una relación R se descompone en fragmentos F_R=$\{fr_1, fr_2, ..., fr_n\}$, cada elemento de datos que puede encontrarse en R también puede encontrarse en uno o más fragmentos fr_i. Esta propiedad es importante en la fragmentación, ya que asegura que los datos en una relación global se mapean a fragmentos sin ninguna pérdida [34]. En el caso de la fragmentación horizontal, el "elemento" se refiere a la tupla, mientras que en el caso de la fragmentación vertical, se refiere a un atributo.

2. *Reconstrucción.* Si una relación R se divide en fragmentos F_R=$\{fr_1, fr_2, ..., fr_n\}$,

debería ser posible definir un operador relacional $\bigtriangledown$ tal que

$R = \bigtriangledown fr_i, \forall fr_i \in F_R$

El operador $\bigtriangledown$ será diferente para las diferentes formas de fragmentación; sin embargo, es importante identificarlo. La reconstrucción de la relación desde sus fragmentos asegura que se preservan las restricciones definidas en los datos en forma de dependencias.

3. *Disyunción.* Si una relación R se descompone horizontalmente en fragmentos F_R={fr_1, fr_2, ..., fr_n} y un elemento de datos d_i está en fr_j, éste no debe estar en ningún otro fragmento fr_k ($k \neq j$). Este criterio asegura que los fragmentos horizontales sean disjuntos. Si la relación R se descompone verticalmente, sus atributos de clave primaria se repiten en todos sus fragmentos (para reconstruir la relación). Por lo tanto, en el caso de la fragmentación vertical, la disyunción se define sólo en los atributos que nos son clave primaria de una relación.

Alternativas de asignación

Asumiendo que la base de datos se fragmentó apropiadamente, entonces se debe decidir cómo asignar los fragmentos a los diversos sitios de la red. Cuando se asignan los datos, éstos pueden replicarse o mantenerse como una sola copia. Las razones para la replicación son la fiabilidad y eficiencia de las consultas de sólo lectura. Si hay múltiples copias de un elemento de datos, hay muchas oportunidades de que algunas copias de los datos sean accesibles desde algún lugar incluso cuando ocurren fallos en el sistema. Además, las consultas de sólo lectura que acceden a los mismos elementos de datos pueden ejecutarse en paralelo debido a que existen copias en varios sitios. Por otro lado, la ejecución de las consultas de actualización causa problemas, ya que el sistema tiene que asegurar que todas las copias de los datos se actualicen apropiadamente. Por lo tanto, la decisión sobre la replicación es un compromiso que depende de la proporción de las consultas de sólo lectura comparada con las consultas de actualización. Esta decisión afecta casi a todos los algoritmos y las

Tabla 2.1: Comparación de las alternativas de replicación.

	Replicación total	**Replicación parcial**	**Fragmentación**
PROCESAMIENTO DE CONSULTAS	Fácil	Misma	dificultad
GESTIÓN DEL DIRECTORIO	Fácil o inexistente	Misma	dificultad
CONTROL DE CONCURRENCIA	Moderada	Difícil	Fácil
FIABILIDAD	Muy alta	Alta	Baja
REALIDAD	Posible aplicación	Realista	Posible aplicación

funciones de control del DBMS distribuido.

Una base de datos no replicada (frecuentemente llamada base de datos *particionada*), contiene fragmentos que son asignados a sitios y sólo hay una copia de cualquier fragmento en la red. En el caso de la replicación, la base de datos puede existir en su totalidad en cada sitio (base de datos *completamente replicada*) o los fragmentos se distribuyen a los sitios de tal forma que las copias de un fragmento pueden residir en múltiples sitios (base de datos *parcialmente replicada*). En la última el número de copias de un fragmento puede ser una entrada para el algoritmo de asignación o la variable de decisión cuyo valor se determina por el algoritmo. La Tabla 2.1 compara las tres alternativas de replicación con respecto a varias funciones del DBMS distribuido [11].

Requerimientos de información

Un aspecto del diseño de la distribución es que demasiados factores contribuyen para un diseño óptimo. La organización lógica de la base de datos, la ubicación de las aplicaciones, las características de acceso de las aplicaciones a la base de datos y las propiedades de los sistemas de cómputo en cada sitio influyen en las decisiones de distribución. Esto hace muy complicado formular el problema de la distribución.

La información necesaria para realizar el diseño de la distribución puede dividirse en cuatro categorías: información de la base de datos, información de las aplicaciones, información de comunicación a través de la red e información del sistema de cómputo. Las últimas

dos categorías son completamente cuantitativas en naturaleza y se usan en los modelos de asignación pero no en los algoritmos de fragmentación.

2.2.4. La complejidad de los problemas

La fragmentación y la asignación en bases de datos distribuidas son los problemas clave del diseño de la distribución de la base de datos. La investigación en esta área con frecuencia involucra métodos de diseño (por ejemplo, programación matemática) con el fin de minimizar el costo combinado de almacenar la base de datos, procesar consultas y el costo de comunicación [11]. Es prácticamente imposible estudiar el diseño de la distribución junto con otros problemas porque cada uno de los problemas es lo suficientemente difícil para estudiarse por sí mismo.

Está comprobado que el problema combinado de la fragmentación y la asignación es NP-hard [35]. Tanto la fragmentación como la asignación son técnicas de diseño de la distribución que se usan para mejorar el desempeño del sistema. Cada uno de los problemas tiene un inmenso espacio de búsqueda para encontrar las soluciones óptimas.

En el caso de la fragmentación horizontal, si un predicado simple n se considera para ejecutar la fragmentación horizontal, 2^n es el número de fragmentos horizontales que se pueden obtener usando predicados mintérmino (conjunciones de predicados simples). Si hay k nodos en la red, la complejidad de asignar los fragmentos horizontales es $O(k^{2^n})$. Por ejemplo, usando 6 predicados simples para ejecutar la fragmentación horizontal resulta en 2^6=64 fragmentos. Para encontrar la asignación óptima de los fragmentos a 4 nodos deben compararse $4^{64} \approx 10^{39}$ posibles asignaciones. Esto es prácticamente incalculable con el poder de las computadoras actuales.

En el caso de la fragmentación vertical, si una relación tiene m atributos que no son llave primaria, los fragmentos posibles que se pueden obtener están dados por el número de Bell que es aproximadamente $B(m) \approx m^m$. Con este número de posibles fragmentos, la asignación de fragmentos usando un modelo de costo apropiado es de la complejidad de $O(k^{m^m})$, con k como el número de nodos de la red. Se proponen en la literatura técnicas

heurísticas para la fragmentación vertical con el objetivo de reducir la complejidad. Por ejemplo, la técnica basada en afinidad que usa una función objetivo propuesta en [36] es de complejidad de $O(m^2 \cdot \log m)$, mientras que el método gráfico propuesto en [37] es de complejidad $O(m^2)$.

No es computacionalmente factible usar un modelo de costo para evaluar todos los posibles esquemas de fragmentación que resultan de usar predicados mintérmino o todos los esquemas de fragmentación vertical posibles. De hecho, la utilización de la afinidad para agrupar atributos o predicados puede reducir el número de fragmentos en el resultado del esquema de fragmentación. Sin embargo, debido a los problemas de las propuestas basadas en afinidad para la fragmentación vertical y la fragmentación horizontal, se reduce la posibilidad de obtener un diseño de distribución óptimo en la fase de asignación.

Debido a la complejidad de los problemas de fragmentación y asignación, con frecuencia se trata a la asignación independientemente de la fragmentación. En la literatura, la mayoría de las propuestas de asignación asumen que ya se realizó la fragmentación. La salida de la fragmentación es la entrada de la asignación. La motivación para aislar el problema de la asignación es para simplificar la formulación del problema por medio de la reducción del espacio de búsqueda [13].

2.3. Procesamiento de consultas

El éxito de la tecnología de bases de datos relacionales en el procesamiento de datos se debe, en parte, a la disponibilidad de los lenguajes no procedimentales (esto es, SQL), el cual puede mejorar significativamente el desarrollo de aplicaciones y la productividad del usuario final. Los lenguajes de bases de datos relacionales de alto nivel permiten que la expresión de consultas complejas se realice de una forma concisa y simple, esto se debe a que esconden los detalles de bajo nivel acerca de la organización física de los datos. En particular, para construir la respuesta a la consulta, el usuario no especifica precisamente el proceso a seguir. Este procedimiento es ideado por un módulo del DBMS, usualmente llamado procesador de consultas. Este módulo libera al usuario de la optimización de

consultas, una tarea que consume mucho tiempo y que es mejor manejada por el procesador de consultas, ya que puede explotar una gran cantidad de información útil acerca de los datos.

Debido a que es un problema de desempeño crítico, el procesamiento de consultas ha recibido (y continúa recibiendo) atención considerable tanto en el contexto de DBMS centralizado como en el distribuido. Sin embargo, el problema de procesamiento de consultas es mucho más difícil en ambientes distribuidos que en centralizados, debido a que un gran número de factores afecta el desempeño de las consultas distribuidas. En particular, las relaciones involucradas en las consultas distribuidas pueden estar fragmentadas y/o replicadas, esto ocasiona costos de comunicación. Además, con muchos sitios que acceder, el tiempo de respuesta de las consultas puede volverse muy alto [11].

2.3.1. El problema del procesamiento de consultas

La principal función del procesador de consultas relacionales es transformar una consulta de alto nivel (normalmente en cálculo relacional) en una consulta de bajo nivel equivalente (típicamente, en alguna variación del álgebra relacional). La consulta de bajo nivel realmente implementa la estrategia de ejecución para la consulta. La transformación debe ser correcta y eficiente. Es correcta si la consulta de bajo nivel tiene el mismo significado que la consulta original, esto es, si ambas consultas producen el mismo resultado. Debido al mapeo bien definido del cálculo relacional al álgebra relacional el problema de la transformación correcta se resuelve fácilmente. Pero producir una estrategia de ejecución eficiente es una tarea más complicada. Una consulta en cálculo relacional puede tener muchas transformaciones equivalentes y correctas en álgebra relacional. Debido a que cada estrategia de ejecución equivalente puede guiar a consumos de recursos de cómputo muy diferentes, la principal dificultad es seleccionar la estrategia de ejecución que minimice el consumo de recursos.

Ejemplo 2.1 *Considerando el siguiente subconjunto del esquema de la base de datos de la compañía constructora visto anteriormente:*

EMP(**ENO**, *ENOMBRE, PUESTO*)

ASG(**ENO**, ***PNO***, *RESP, DUR*)

y la siguiente consulta:

"Encuentra los nombres de los empleados que administran un proyecto"

La expresión de la consulta en cálculo relacional usando la sintaxis SQL es:

```
SELECT ENOMBRE
FROM EMP, ASG
WHERE EMP.ENO=ASG.ENO
AND RESP='Administrador'.
```

Dos consultas equivalentes en el álgebra relacional que son transformaciones correctas de las consultas anteriores son:

$\Pi_{ENAME}(\sigma_{RESP=\text{`}Administrador\text{'}\wedge EMP.ENO=ASG.ENO}(EMP \times ASG))$

y

$\Pi_{ENAME}(EMP * (\sigma_{RESP=\text{`}Administrador\text{'}}(ASG)))$

La segunda consulta evita el producto cartesiano de **EMP** y **ASG**, por lo tanto, consume mucho menos recursos de cómputo que la primera, así que esta debe retenerse.

En un contexto centralizado, las estrategias de ejecución de consultas pueden expresarse bien usando álgebra relacional. El principal rol del procesador de consultas centralizado es elegir, para una consulta dada, la mejor consulta en álgebra relacional entre todas las equivalentes. Debido a que el problema es computacionalmente intratable con un gran número de relaciones [38], generalmente se reduce a elegir una solución cercana a la óptima.

En un sistema distribuido, el álgebra relacional no es suficiente para expresar estrategias de ejecución. Así que debe complementarse con operadores para intercambiar datos entre sitios. Además de la elección de la ordenación de los operadores del álgebra relacional, el procesador de consultas distribuido debe detectar también los mejores sitios para procesar los datos, y posiblemente la forma en que deben transformarse los datos. Esto incrementa el espacio de solución para elegir la estrategia de ejecución distribuida, haciendo el procesamiento de consultas distribuidas significativamente más difícil.

Ejemplo 2.2 *Este ejemplo ilustra la importancia de la selección y comunicación de sitios para la elección de una consulta en álgebra relacional ejecutada en una base de datos fragmentada. Considerando la siguiente consulta de ejemplo:*

$\Pi_{ENAME}(EMP * (\sigma_{RESP='Administrador'}(ASG)))$

Asumiendo que la relación ***EMP*** *y* ***ASG*** *se fragmentan horizontalmente como sigue:*

$EMP_1 = \sigma_{ENO \leq 'E3'}(EMP)$

$EMP_2 = \sigma_{ENO > 'E3'}(EMP)$

$ASG_1 = \sigma_{ENO \leq 'E3'}(ASG)$

$ASG_2 = \sigma_{ENO > 'E3'}(ASG)$

Los fragmentos **ASG1, ASG2, EMP1 y EMP2** se almacenan en los sitios 1, 2, 3 y 4 respectivamente, y el resultado se espera en el sitio 5. Dos estrategias de ejecución distribuidas equivalentes para la consulta anterior se muestran en las Figuras 2.8 y 2.9. Una flecha del sitio i al sitio j etiquetada con **R** indica que la relación **R** se transfiere del sitio i al sitio j. La estrategia A explota el hecho de que las relaciones **EMP** y **ASG** se fragmentan de la misma forma con el fin de ejecutar el operador select y join en paralelo. La estrategia B centraliza todos los operandos de datos en el sitio del resultado antes de procesar la consulta.

Para evaluar el consumo de recursos de estas dos estrategias se usa un modelo de costo simple. Se asume que un acceso a una tupla *tupacc*, es 1 unidad y una transferencia de una tupla denotada *tuptrans*, es 10 unidades. Se asume que las relaciones **EMP** y **ASG** tienen 400 y 1000 tuplas, respectivamente, y que hay 20 administradores en la relación **ASG**. También se supone que los datos se distribuyen uniformemente entre los sitios. Finalmente, se asume que las relaciones **ASG** y **EMP** se agrupan localmente en los atributos *RESP* y *ENO*, respectivamente. Por lo tanto, hay un acceso directo a las tuplas de **ASG** con base en el valor del atributo *RESP* y a las tuplas de **EMP** con base en el atributo *ENO*. El costo total de la estrategia A puede derivarse como sigue:

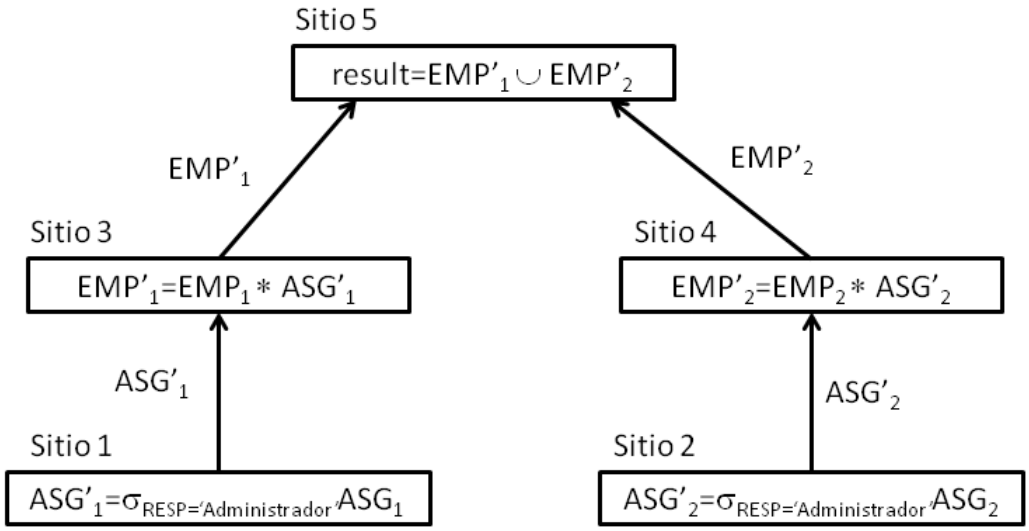


Figura 2.8: Estrategia A

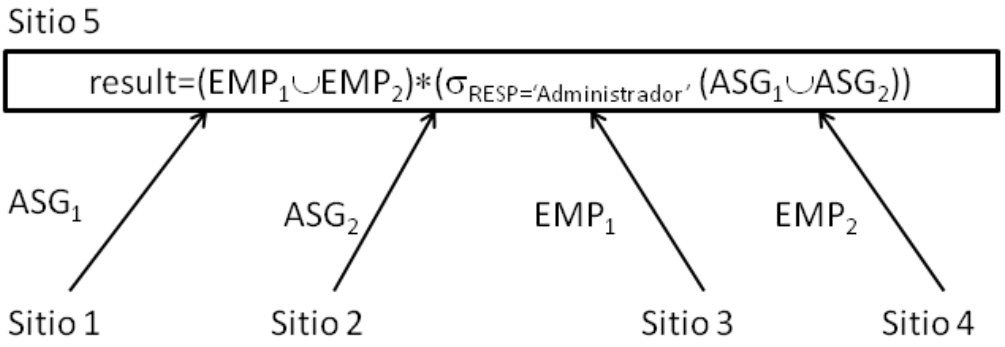


Figura 2.9: Estrategia B

1.	Producir **ASG'** por medio de la selección de **ASG** requiere (10+10)**tupacc*	=	20
2.	Transferir **ASG'** a los sitios de **EMP** requiere (10+10)**tuptrans*	=	200
3.	Producir **EMP'** por medio de la reunión entre **ASG'** y **EMP** requiere (10+10)**tupacc**2	=	40
4.	Transferir **EMP'** al sitio del resultado requiere (10+10)**tuptrans*	=	200
	El costo total es		460

El costo de la estrategia B puede derivarse como sigue:

1.	Transferir **EMP** al sitio 5 requiere 400**tuptrans*	=	4,000
2.	Transferir **ASG** al sitio 5 requeire 1000**tuptrans*	=	10,000
3.	Producir **ASG'** por medio de la selección de **ASG** requiere 1000**tuptrans*	=	1,000
4	Reunir **EMP** y **ASG'** requiere 400*20**tupacc*	=	8,000
	El costo total es		23,000

En la estrategia A, la reunión de **ASG**' y **EMP** (paso 3) puede explotar el índice agrupado en *ENO* de **EMP**. Por lo tanto, **EMP** se accede sólo una vez por cada tupla de **ASG**'. En la estrategia B, se asume que los métodos de acceso a las relaciones **EMP** y **ASG** que se basan en los atributos *RESP* y *ENO* se pierden debido a la transferencia de datos. Esta es una suposición razonable en la práctica. Se asume que la reunión de **EMP** y **ASG**' en el paso 4 se hace por medio del algoritmo de ciclo agrupado (que simplemente ejecuta el producto cartesiano de las dos relaciones de entrada). La estrategia A es mejor por un factor de 50, lo cual es bastante significativo. Además, se provee mejor distribución de trabajo entre los sitios. La diferencia sería mucho mayor si se asumiera una comunicación más lenta y/o un mayor grado de fragmentación.

2.3.2. Costos en el procesamiento de consultas

El objetivo del procesamiento de consultas en un ambiente distribuido es transformar una consulta de alto nivel en una base de datos distribuida, la cual es vista por los usuarios

como una sola base de datos, en una estrategia de ejecución eficiente expresada en un lenguaje de bajo nivel en las bases de datos locales. Se asume que el lenguaje de bajo nivel es el cálculo relacional, mientras que el lenguaje de alto nivel es una extensión del álgebra relacional con operadores de comunicación. Un aspecto importante del procesamiento de consultas es la optimización de consultas. Debido a que muchas estrategias de ejecución son transformaciones correctas de la misma consulta de alto nivel, aquella que optimice (minimice) el consumo de recursos debe retenerse.

Una buena medida para el consumo de recursos es el costo total que se incurre en el procesamiento de la consulta [39]. El costo total es la suma de todos los tiempos incurridos en el procesamiento de los operadores de la consulta en varios sitios y en la comunicación entre sitios. Otra buena medida es el tiempo de respuesta de la consulta que es el tiempo que tarda la ejecución de una consulta [40]. Debido a que los operadores pueden ejecutarse en paralelo en diferentes sitios, el tiempo de respuesta de la consulta puede ser significativamente menor que su costo total.

En un sistema distribuido de bases de datos, el costo total que debe minimizarse incluye costos de CPU, E/S y comunicación. El costo de CPU se incurre cuando se ejecutan operadores sobre los datos en la memoria principal. El costo de E/S es el tiempo necesario para accesos a disco. Este costo puede minimizarse reduciendo el número de accesos a disco por medio de métodos de acceso rápido a los datos y el uso eficiente de la memoria principal (gestión de la memoria intermedia). El costo de comunicación es el tiempo necesario para intercambiar datos entre los sitios que participan en la ejecución de la consulta. Este costo se ocasiona en el procesamiento de mensajes y en la transmisión de datos a través de la red de comunicación.

Los primeros dos componentes (costo de E/S y costo de CPU) son los únicos factores considerados por DBMSs centralizados. El componente de costo de comunicación es un factor igualmente importante considerado en las bases de datos distribuidas. La mayoría de las primeras propuestas para optimización de consultas distribuidas asumen que el costo de comunicación domina ampliamente el costo de procesamiento local (costo de E/S y CPU)

y, por lo tanto, no lo toman en cuenta. Esta suposición se basa en redes de comunicación muy lentas (por ejemplo, las redes de área extendida que tenían un ancho de banda de unos cuantos kilobytes por segundo), en vez de basarse en redes con ancho de banda que son comparables al ancho de banda de conexión al disco. Por lo tanto, el objetivo del procesamiento de consultas distribuidas se reduce al problema de minimizar costos de comunicación descartando el procesamiento local. La ventaja es que la optimización local puede hacerse independientemente usando los métodos conocidos para bases de datos centralizadas. Sin embargo, los ambientes de procesamiento distribuidos modernos tienen redes de comunicación mucho más rápidas, cuyo ancho de banda es comparable al de los discos. Por lo tanto, las propuestas más recientes consideran una combinación de peso de estos tres componentes de costo debido a que los tres contribuyen al costo total de evaluar una consulta [41]. Sin embargo, en ambientes distribuidos con ancho de banda elevado, los costos generales incurridos en la comunicación entre los sitios (por ejemplo, protocolos de software) hacen que el costo de comunicación todavía sea un factor importante [11].

En sistemas de bases de datos muy grandes, el costo de acceso a datos desde el disco es normalmente el costo más importante, ya que los accesos a disco son lentos comparados con las operaciones en la memoria. Además, las velocidades del CPU han mejorado mucho más rápido que las velocidades del disco. Por lo tanto, es probable que el tiempo gastado en las actividades del disco siga influenciando el tiempo total de ejecución de una consulta. Por último, las estimaciones del tiempo de CPU son más difíciles de hacer, comparadas con las estimaciones del costo de acceso a disco. Por este motivo se considera el costo de los accesos a disco una medida razonable del costo de procesamiento de una consulta [2].

2.4. Comentarios

Esta tesis se enfoca en el problema de la fragmentación vertical dentro del diseño de bases de datos distribuidas. En este Capítulo se presentaron los conceptos básicos de las bases de datos distribuidas, su evolución y ventajas, así como las complicaciones introducidas por la distribución, la que más nos interesa es el diseño de la base de datos distribuida.

Se explicaron las dos estrategias de diseño: descendente y ascendente. La estrategia descendente consiste en generar esquemas conceptuales locales (fragmentos) de un esquema conceptual global (tabla). En el siguiente Capítulo se presentan algunos métodos de fragmentación propuestos en la literatura.

Capítulo 3

Métodos de fragmentación

En un DBMS distribuido, las relaciones se almacenan a través de varios sitios de una red de computadoras. Acceder a una relación que esta almacenada en un sitio remoto incurre en costos de comunicación (costos de transferencia o transporte), para reducir estos costos, una relación puede fragmentarse a través de varios sitios, y los fragmentos pueden almacenarse en los sitios donde se acceden con mayor frecuencia [42].

La fragmentación es una técnica de diseño para dividir una relación de una base de datos en dos o más particiones de tal forma que la combinación de las particiones produzca la relación original sin ninguna pérdida o adición de información [13].

El problema de la fragmentación de datos apareció antes de las bases de datos distribuidas, incluso antes del surgimiento de las bases de datos, para reducir los costos de almacenamiento y de transferencia en los sistemas de archivos [43], [44]. Los primeros trabajos relacionados con la fragmentación en bases de datos distribuidas son [45], [46], [36]. Con el surgimiento del modelo de datos orientado a objetos, las propuestas de fragmentación existentes se adoptaron en el modelo de datos orientado a objetos, tomando en consideración las características de dicho modelo. La fragmentación de datos permite la ejecución paralela de una consulta simple, reduce la cantidad de accesos a datos irrelevantes y la transferencia de datos innecesaria, incrementa el nivel de concurrencia y, por lo tanto, la capacidad de procesamiento en un sistema distribuido de bases de datos [11].

Si la relación R se fragmenta, R se divide en varios fragmentos fr_1, fr_2, ..., fr_n. Estos fragmentos contienen suficiente información como para permitir la reconstrucción de la relación original R. Hay dos esquemas diferentes de fragmentación de las relaciones: fragmentación horizontal y fragmentación vertical. La combinación de estos dos esquemas da origen a un tercer esquema: la fragmentación híbrida. La fragmentación puede ser estática o dinámica [47]. Las técnicas de fragmentación estáticas obtienen un esquema de fragmentación para una carga de trabajo esperada. Esta carga de trabajo es un conjunto de consultas de la base de datos obtenidas por el DBA (administrador de la base de datos). Las técnicas de fragmentación dinámicas monitorizan continuamente la base de datos y adaptan el esquema de fragmentación a la carga de trabajo en todo momento. Las técnicas dinámicas son parte de la tendencia hacia un ajuste completamente automático [48], el cual se ha vuelto una línea de investigación popular [14]. En este Capítulo se presenta una clasificación de las técnicas de fragmentación en cuatro grupos: fragmentación estática en bases de datos tradicionales (bases de datos relacionales u orientadas a objetos que sólo contienen datos alfanuméricos), fragmentación estática en bases de datos multimedia, fragmentación dinámica en bases de datos tradicionales y fragmentación dinámica en bases de datos multimedia.

3.1. Fragmentación estática en bases de datos tradicionales

3.1.1. Fragmentación horizontal

En la fragmentación horizontal la relación R se divide en varios fragmentos fr_1, fr_2, ..., fr_n. Cada tupla de la relación R debe pertenecer como mínimo a uno de los fragmentos, de modo que se pueda reconstruir la relación original si fuera necesario.

Un fragmento horizontal se produce especificando un predicado que ejecuta una restricción en las tuplas de una relación. El predicado se define usando la operación de selección del álgebra relacional.

Definición 3.1 *Para una relación dada R, un fragmento horizontal se define como*

$$fr_i = \sigma_{p_k}(R)$$

donde p_k es un predicado basado en uno o más atributos de la relación R [49].

Si tenemos la relación EQUIPO correspondiente a los equipos de una compañía de venta de maquinaria de la Figura 3.1. La Figura 3.2 muestra la relación EQUIPO de la Figura 3.1 dividida horizontalmente en dos fragmentos. El fragmento EQUIPO_1 contiene información acerca de los equipos cuyo nombre es igual a 'Motobomba', mientras que EQUIPO_2 almacena información acerca de los equipos con nombre igual a 'Podadora'.

Los fragmentos de la Figura 3.2 pueden definirse de la siguiente forma:

$$EQUIPO_1 = \sigma_{NOMBRE='Motobomba'}(EQUIPO)$$

$$EQUIPO_2 = \sigma_{NOMBRE='Podadora'}(EQUIPO)$$

Se reconstruye la relación R tomando la unión de todos los fragmentos, es decir,

$$R = fr_1 \cup fr_2 \cup ... \cup fr_n$$

La fragmentación horizontal suele utilizarse para conservar las tuplas en los sitios en que más se utilizan, y de esta manera minimizar la transferencia de datos [2].

EQUIPO

ID	NOMBRE	MCA	MOD	CAB	PRECIO	CANT
MB1	Motobomba	Super	MT2AMD	8	5000	20
MB2	Motobomba	Gimexsa	GBG-20	5.5	3500	15
PO1	Podadora	Rally	TVS90	3.8	2800	12
PO2	Podadora	Truper	P-2240	4	3000	10

Figura 3.1: Relación EQUIPO.

$EQUIPO_1$

ID	NOMBRE	MCA	MOD	CAB	PRECIO	CANT
MB1	Motobomba	Super	MT2AMD	8	5000	20
MB2	Motobomba	Gimexsa	GBG-20	5.5	3500	15

$EQUIPO_2$

ID	NOMBRE	MCA	MOD	CAB	PRECIO	CANT
PO1	Podadora	Rally	TVS90	3.8	2800	12
PO2	Podadora	Truper	P-2240	4	3000	10

Figura 3.2: Ejemplo de fragmentación horizontal.

Fragmentación horizontal en bases de datos relacionales

Varios investigadores han trabajado en fragmentación horizontal en el modelo de datos relacional.

Ceri, Negri y Pellagati [50] fragmentan archivos horizontalmente utilizando predicados mintérmino (ver Definición 3.3) para optimizar el número de accesos ejecutados por los programas de aplicación a diferentes porciones de los datos. Ellos declaran que los fragmentos mintérmino contienen registros que las consultas acceden de manera homogénea y, por lo tanto, son las unidades óptimas de asignación.

Definición 3.2 *Para una relación dada R= {a_1:D_1,...,a_n:D_n}, un predicado simple es de la forma*

$$p_k : a_i \theta Valor$$

Donde a_i es un atributo definido sobre un dominio D_i, $\theta \in \{=, <, \neq, \leq, >, \geq\}$ y Valor $\in$ D_i. Un conjunto de predicados simples definidos en una relación R se denota por Pr={p_1, p_2, ..., p_m} [11].

Definición 3.3 *Los predicados mintérmino M={m_1, m_2,..., m_z} sobre un conjunto Pr de predicados simples son las conjunciones de los predicados simples y sus negaciones:*

$$M = \{m_j | m_j = \bigwedge_{p_k \in P_r} \overset{*}{p_k}\}, k = 1, ..., m, j = 1, ..., z.$$

Donde $\overset{*}{p_k} = p_k$ *o* $\overset{*}{p_k} = \neg p_k$*. Todos los predicados simples en Pr aparecen en cada predicado mintérmino, ya sea positiva o negativamente [11].*

Definición 3.4 *Se dice que un predicado simple es completo si y sólo si hay una probabilidad igual de que cada consulta acceda cualquier tupla que corresponda a cualquier fragmento que se define de acuerdo a Pr.*

Definición 3.5 *Suponiendo que* m_i *contiene a* p_k*, y supongamos que* m_j *es obtenida de* m_i *reemplazando* p_k *por* $\neg p_k$*. Supongamos que* fr_i *y* fr_j *son los fragmentos definidos de acuerdo a* m_i *y a* m_j *respectivamente. Entonces* p_k *es relevante si y sólo si*

$$\frac{acc(m_i)}{card(fr_i)} \neq \frac{acc(m_j)}{card(fr_j)}$$

Donde acc(m_i*) y acc(*m_j*) denota la frecuencia de acceso de los predicados mintérmino* m_i *y* m_j*, card(*fr_i*) y card(*fr_j*) denota las cardinalidades de los fragmentos* fr_i *y* fr_j*. Si todos los predicados de un conjunto Pr son relevantes entonces Pr es mínimo [13].*

Özsu y Valduriez [11] definen la información de la base de datos (esquema conceptual global, enlaces entre relaciones, cardinalidad de las relaciones) y de las consultas (predicados simples usados en las consultas, predicados mintérmino, frecuencia de accesos de las consultas) necesaria para realizar la fragmentación horizontal y presentaron un algoritmo iterativo COM_MIN, para generar un conjunto completo y mínimo de predicados *Pr'* de un conjunto de predicados simples dado *Pr*. El algoritmo verifica cada predicado p_k en el conjunto de predicados simples *Pr* dado para ver si puede usarse para fragmentar la relación R en al menos dos partes que se accedan de manera diferente por al menos una consulta. Si p_k satisface la regla fundamental de completitud y minimalidad entonces debería incluirse en *Pr'*. Si no es relevante, entonces p_k debería removerse de *Pr'*. Presentan

también un algoritmo llamado PHORIZONTAL para describir la fragmentación horizontal primaria. Este usa el algoritmo COM_MIN y un conjunto de implicaciones I como entrada para producir un conjunto de predicados mintérmino satisfactorios M. Si un predicado mintérmino m_i es contradictorio a una regla de implicación en I, entonces se remueve de M. Los fragmentos mintérmino se definen de acuerdo al conjunto de predicados mintérmino satisfactorios M. Pero el conjunto de implicaciones I es difícil de definir.

De hecho, el algoritmo no es muy práctico, ya que siempre resultará en un conjunto Pr' de Pr, el conjunto de predicados mintérmino M' determinado por Pr' y el conjunto correspondiente de fragmentos. Los predicados simples se omiten de Pr si no contribuyen a la fragmentación, esto es, ellos sólo violan el principio de minimalidad. Esto resulta en considerar sólo los predicados simples en las consultas más importantes y tomar todos los predicados mintérmino satisfactorios. Esto obviamente guía a fragmentos que son accedidos de manera diferente por al menos dos consultas. El algoritmo posteriormente no da reglas ejecutables para eliminar los predicados mintérmino insatisfactorios. El principal problema, sin embargo, es que el número de fragmentos resultante del algoritmo es exponencial en el tamaño de Pr. En la práctica, sería importante reducir este número significativamente, lo cual significaría recombinar algunos de los fragmentos. De hecho esto implica dejar el principio de completitud y reemplazarlo por el criterio de optimización basado en un modelo de costo [13].

Shin e Irani [51] dividen relaciones horizontalmente con base en grupos de referencia de usuarios (URCs, en inglés: User Reference Clusters). Los URCs se estiman de las consultas de los usuarios pero se refinan usando conocimiento semántico de las relaciones. Considerando que el propósito de la fragmentación horizontal es minimizar el número total de accesos a disco, Khalil et al. [52] presentan un algoritmo de fragmentación horizontal basado en consultas, que toma como entrada una matriz de uso de predicados. Sin embargo, la meta de la fragmentación en bases de datos distribuidas no es sólo reducir el número total de accesos a disco, sino también los costos de transporte de datos, que dominan el costo total de las consultas.

Dimovski et al. [53] proponen un método de fragmentación horizontal para encontrar un diseño adecuado de un datawarehouse relacional con el fin de optimizar las consultas. Este método tiene cuatro pasos: en el primer paso se extraen todos los predicados usados en las consultas realizadas en el datawarehouse, después se encuentra un conjunto de predicados completo correspondiente a cada relación dimensión, posteriormente se usa un algoritmo llamado *ComputeMin* que encuentra un conjunto mínimo de predicados para cada relación y finalmente aplican un algoritmo genético para encontrar un esquema de fragmentación óptimo.

Fragmentación horizontal en bases de datos orientadas a objetos

En [54, 55], los autores desarrollaron esquemas de fragmentación horizontal y presentaron una solución para soportar transparencia de métodos en bases de datos orientadas a objetos (BDOOs). Ezeife et al. [56] desarrollaron un conjunto de algoritmos para fragmentar horizontalmente cuatro modelos de clases: con atributos y métodos simples, con atributos complejos y métodos simples, con atributos simples y métodos complejos y con atributos y métodos complejos. Ellos usan el algoritmo desarrollado en [11]. Sin embargo, el número de predicados mintérmino generados es exponencial en el número de predicados. Bellatreche et al. [57] presentan la fragmentación horizontal como un proceso para reducir el número de accesos a disco al ejecutar consultas por medio de la reducción del número de accesos a objetos irrelevantes. Se extienden los predicados de las consultas del modelo relacional a los predicados definidos en los métodos. Dichos predicados se agrupan de acuerdo a una matriz de afinidad de predicados.

Definición 3.6 *La afinidad de predicados entre cada par de predicados simples p_i, p_j es la suma de las frecuencias de las consultas que acceden ambos predicados juntos [13].*

Los grupos de predicados resultantes son de la misma longitud que el número de predicados simples y se definen en el mismo conjunto de atributos y métodos.

En [58] presentan tres algoritmos de fragmentación horizontal en BDOOs: el primero se basa en un modelo de costo que tiene como objetivo reducir el número de accesos a disco en la ejecución de las consultas, el segundo se basa en afinidad de predicados y utiliza el algoritmo gráfico de Navathe y Ra [37] para obtener los fragmentos, y el último es un algoritmo aproximado que se basa en una técnica de ascenso de colinas cuyo estado inicial se obtiene aplicando el algoritmo de afinidad y posteriormente se utiliza el modelo de costo para encontrar el esquema de fragmentación horizontal con el menor costo. Sin embargo, el modelo de costo no toma en cuenta el costo de transporte de datos, ya que sólo se consideran bases de datos centralizadas, por tal motivo estos algoritmos no pueden aplicarse en bases de datos distribuidas donde el costo de transporte domina el costo de ejecución de las consultas. Además la complejidad del algoritmo basado en costo es muy alta debido a que se basa en un procedimiento de enumeración exhaustiva.

Ezeife y Zheng [59] propusieron un evaluador de particiones horizontales de objetos (OHPE, en inglés: Object Horizontal Partition Evaluator), que contiene dos componentes: el costo de acceso irrelevante local y el costo de acceso relevante remoto. Sin embargo, ambos componentes sólo miden el número de accesos a los fragmentos y no toman en cuenta el tamaño de los objetos.

Baiao et al. [60] extienden el algoritmo propuesto en [61] y toman la matriz de afinidad de predicados como entrada para construir un grafo de afinidad de predicados que se usa para definir los fragmentos horizontales. La desventaja es que el esquema de fragmentación resultante sólo toma en cuenta la frecuencia de acceso a los datos pero no considera el tamaño de los objetos. Cheng et al. [62] también toma la afinidad de predicados como entrada para realizar la fragmentación horizontal por medio de una agrupación de los predicados de acuerdo a la distancia entre ellos y formulan el problema de fragmentación horizontal como el problema del agente viajero (TSP, en inglés: Traveling Salesman Problem).

Darabant et al. [63] proponen un método para fragmentación horizontal de clases con atributos y métodos simples y con atributos y métodos complejos [64] que usa el método

de agrupamiento *k-means* para dividir objetos en fragmentos.

Ventajas y desventajas de la fragmentación horizontal

La fragmentación horizontal comúnmente se utiliza para obtener un mejor desempeño tanto en bases de datos relacionales como en bases de datos orientadas a objetos por medio de la reducción de los accesos a disco requeridos para ejecutar consultas y la transferencia de datos entre sitios.

Actualmente, los DBAs usan extensamente la fragmentación horizontal para hacer los servidores de la base de datos fáciles de gestionar. Si la tabla se fragmenta, las operaciones como respaldar o restablecer la base de datos se vuelven mucho más fáciles [65].

Sin embargo, se ha demostrado que el problema de la fragmentación horizontal es NP-hard, por lo que no es posible encontrar una solución factible del problema, en una cantidad razonable de tiempo sin el uso de procedimientos heurísticos [66].

3.1.2. Fragmentación vertical

La fragmentación vertical de una relación R produce los fragmentos fr_1, fr_2, ..., fr_n, cada uno de los cuales contiene un subconjunto de los atributos de R, así como la clave primaria de R.

Definición 3.7 *Un fragmento vertical se define usando la operación de proyección de álgebra relacional. Para una relación dada R, un fragmento vertical se define como*

$$fr_i = \prod_{a_1,a_2,\ldots,a_n}(R)$$

donde a_1, a_2, ..., a_n son atributos de la relación R [49].

La Figura 3.3 muestra la relación de la Figura 3.1 dividida verticalmente en dos fragmentos EQUIPO_1 y EQUIPO_2. EQUIPO_1 contiene sólo la información acerca de los nombres, marcas, modelos y caballaje de los equipos, mientras que EQUIPO_2 contiene los precios y la cantidad de los equipos.

$EQUIPO_1$

ID	NOMBRE	MCA	MOD	CAB
MB1	Motobomba	Super	MT2AMD	8
MB2	Motobomba	Gimexsa	GBG-20	5.5
PO1	Podadora	Rally	TVS90	3.8
PO2	Podadora	Truper	P-2240	4

$EQUIPO_2$

ID	PRECIO	CANT
MB1	5000	20
MB2	3500	15
PO1	2800	12
PO2	3000	10

Figura 3.3: Ejemplo de fragmentación vertical.

Los fragmentos de la Figura 3.3 pueden definirse de la siguiente forma:

$$EQUIPO_1 = \prod_{ID,NOMBRE,MCA,MOD,CAB}(EQUIPO)$$

$$EQUIPO_2 = \prod_{ID,PRECIO,CANT}(EQUIPO)$$

El objetivo de la fragmentación vertical es dividir una relación en un conjunto de relaciones más pequeñas para que muchas de las consultas se ejecuten en sólo un fragmento. En este contexto, una fragmentación "óptima" es aquella que produce un esquema de fragmentación que minimice el tiempo de ejecución de las consultas que accedan atributos de esos fragmentos [11].

Una formulación general del problema de fragmentación vertical es el siguiente: Dada una relación R formada por un conjunto de atributos R=$\{a_1, \ldots, a_n\}$ y un conjunto de consultas Q=$\{q_1,\ldots,q_m\}$ determinar un esquema de fragmentación vertical formado por conjunto de fragmentos EFV=$\{fr_1,\ldots,fr_p\}$ tal que:

1. Cada fragmento $fr \in EFV$ almacene un subconjunto de los atributos de R más la clave primaria de R.

2. Cada atributo $a \in R$, se encuentre en exactamente un fragmento $fr \in EFV$, con excepción de la clave primaria.

3. Minimice la suma de los costos de las consultas cuando se ejecutan en un esquema

de fragmentación vertical $\sum_{q \in Q} Costo(q, EFV)$ [67].

La fragmentación vertical se ha investigado en el contexto de sistemas centralizados de base de datos, así como en los distribuidos. Su motivación en el contexto centralizado es que es una herramienta de diseño, que permite a las consultas tratar con relaciones más pequeñas, y esto causa un menor número de accesos a páginas [36].

Complejidad del problema de fragmentación vertical

La fragmentación vertical es inherentemente más complicada que la fragmentación horizontal, ya que existe un gran número de alternativas para realizarla. Por ejemplo, en la fragmentación horizontal, si el número total de predicados simples p_k es n, hay 2^n posibles predicados mintérmino (conjunción de predicados simples) que pueden definirse en él. Además, algunos de estos contradirán las implicaciones existentes, reduciendo posteriormente los fragmentos candidatos que necesitan considerarse. En el caso de la fragmentación vertical, sin embargo, si una relación tiene m atributos que no sean clave primaria, el número de fragmentos posibles es igual a *B(m)*, que es el m-ésimo número Bell [46]. Para valores grandes de m, $B(m) \approx m^m$; por ejemplo, para m=10, *B(m)*=115975, para m=15, B(m)$\approx 10^9$, para m=30, *B(m)*$\approx 10^{23}$ [45], [36]. Estos valores indican que es inútil intentar obtener soluciones óptimas para el problema de fragmentación vertical; es necesario recurrir a heurísticas. A continuación presentamos las propuestas heurísticas de fragmentación vertical en bases de datos relacionales y orientadas a objetos.

Fragmentación vertical en bases de datos relacionales

Las propuestas heurísticas de fragmentación vertical en bases de datos relacionales se dividen en basadas en afinidad, basadas en costo [13] y basadas en reglas de asociación.

Las propuestas basadas en afinidad toman en cuenta el acceso a dos atributos juntos, mientras más consultas accedan a dos atributos juntos mayor será el valor de afinidad entre esos atributos, por lo tanto, los fragmentos se forman de acuerdo a esos valores de afinidad. Algunas propuestas basadas en afinidad son [68], [36], [69], [37], [70], [62], [71], [72].

La principal desventaja de las propuestas basadas en afinidad es que esta medida no refleja la afinidad cuando se involucran más de dos atributos. Por lo tanto, no es posible medir la afinidad de los fragmentos completos [73]. Las propuestas basadas en costo ofrecen una solución a este problema. En estos métodos se utilizan modelos de costo para obtener los fragmentos verticales. Ejemplos de este tipo de propuestas incluyen [45], [73], [74], [75], [76], [65].

Recientemente se desarrollaron métodos de fragmentación vertical basados en una técnica de minería de datos conocida como minería de reglas de asociación, estos métodos consisten en buscar conjuntos de atributos que ocurren simultáneamente en una transacción o consulta. Entre estos métodos tenemos [77], [78].

Propuestas basadas en afinidad

Definición 3.8 *Dado un par de atributos a_i, a_j de una relación $R=\{a_1,...,a_n\}$ al cual acceden un conjunto de consultas $Q=\{q_k, ..., q_m\}$, la afinidad se define como*

$$aff(a_i, a_j) = \sum_{k=1|u(q_k,a_i)=1\wedge u(q_k,a_j)=1}^{m} f_k$$

donde f_k es la frecuencia de acceso de la consulta q_k y $u(q_k, a_i)=1$ cuando la consulta q_k usa el atributo a_i. El cálculo de la afinidad para cada par de atributos resulta en una matriz $n \times n$, llamada matriz de afinidad de atributos (AAM).

El primer trabajo relacionado con el uso de afinidad de atributos para realizar la fragmentación vertical fue el de Hoffer y Severance [68], que utilizó el algoritmo de enlace de energía (BEA, en inglés: Bond Energy Algorithm) para agrupar los atributos de acuerdo a su afinidad, el problema que tenía este algoritmo es que el diseñador de la base de datos tenía que determinar como crear las particiones. Por lo tanto, Navathe et al. [36] resuelven ese problema al desarrollar algoritmos para crear fragmentos verticales en tres ambientes distintos: sitio simple, varios niveles de memoria y múltiples sitios, ellos toman en cuenta la fragmentación vertical superpuesta y no superpuesta, así como la fragmentación vertical

binaria y n-aria, su propuesta se divide en dos pasos:

1. En el primer paso se utiliza como entrada una matriz de uso de atributos (AUM, en inglés: Attribute Usage Matrix) para construir una AAM en la que se realiza un agrupamiento y finalmente para obtener los fragmentos utilizan una función objetivo empírica.

2. En el segundo paso los fragmentos se refinan tomando en cuenta factores de costo, que reflejan el ambiente físico de almacenamiento de los fragmentos.

Su complejidad es de O(n^2 log n), donde n es el número de atributos. La desventaja de este trabajo es que para realizar una fragmentación vertical n-aria era necesario utilizar de manera iterativa el algoritmo de fragmentación vertical binaria después de agrupar los atributos, por lo que de un problema surgían dos nuevos problemas y el agrupamiento tenía que realizarse cada vez.

Ceri et al. [69] extiende el trabajo [36] considerándolo una herramienta divide y le agrega una herramienta conquista que toma en cuenta los accesos físicos y las operaciones de la base de datos. Su desventaja es que también se enfoca en la descomposición del proceso de diseño en varios subproblemas y no mejora la complejidad del algoritmo de fragmentación vertical de Navathe et al. [36].

Navathe y Ra [37] proponen un algoritmo de fragmentación vertical gráfico con el que mejoran el trabajo previo en fragmentación vertical. La principal ventaja de este algoritmo es que genera todos los fragmentos en una iteración en un tiempo de O(n^2), otra de sus ventajas es que no requiere una función objetivo para realizar la fragmentación vertical. El algoritmo consiste en generar un grafo de afinidad con base en la AAM, y se forma un árbol de cobertura (spanning tree) linealmente conectado, después de obtener el árbol de cobertura se construyen ciclos de afinidad que son los fragmentos candidatos. La salida del algoritmo es un conjunto de fragmentos verticales de una relación dada. Las desventajas de este algoritmo es que el número de fragmentos resultantes no es igual al número de nodos de la base de datos distribuida, por lo tanto, es necesaria una recombinación de

los fragmentos, además no realizan una evaluación del esquema de fragmentación vertical resultante y de cómo mejora el desempeño del sistema de la base de datos distribuida.

Lin y Zhang [70] consideran que el algoritmo gráfico de Navathe y Ra [37] no resuelve adecuadamente el problema de la fragmentación vertical, ya que la restricción de encontrar ciclos de afinidad es un problema NP-hard, por lo tanto, extienden el algoritmo de Navathe y Ra, ellos proponen buscar un subgrafo en el grafo de afinidad de al menos dos nodos para los que los valores de afinidad sean mayores que los de cada eje incidente, la principal desventaja de este algoritmo es su complejidad de implementación, ya que si se considera una relación con decenas de atributos, la construcción del grafo de afinidad es muy complicada.

Cheng et al. [62] formula el problema de la fragmentación vertical como un problema TSP, para el que se usa una propuesta de agrupamiento basada en búsqueda genética para mejorar el desempeño del sistema, su objetivo es encontrar submatrices en la matriz de atributos y transacciones, donde una transacción en una submatriz sólo acceda atributos en la misma submatriz. Por lo tanto, los atributos en una submatriz forman un fragmento. Esta propuesta se basa en afinidad porque agrupa atributos con base en la distancia entre ellos. La información de los sitios no se toma en cuenta cuando se realiza la fragmentación. El objetivo de esta propuesta es agrupar atributos de tal forma que se minimice la diferencia entre la distancia promedio dentro de los grupos y la distancia promedio entre grupos. Sin embargo, no hay prueba de que esta propuesta pueda minimizar realmente los costos de consulta totales.

Du et al. [79] proponen una *medida de atracción* que se basa en la probabilidad de que dos atributos sean utilizados juntos en una consulta, consideran que esta medida puede evaluar globalmente la afinidad entre los atributos por lo que superan las limitaciones de las propuestas anteriores. Usan la AAM para obtener la matriz de atracción de atributos, la definición de la medida de atracción se muestra a continuación.

Definición 3.9 *El valor de atracción de dos atributos a_i, a_j se define como*

$$att(a_i, a_j) = \frac{aff(a_i, a_j)}{aff(a_i, a_i)}$$

Donde aff(a_i, a_j) es el valor de afinidad de los atributos a_i, a_j. El cálculo de la atracción para cada par de atributos resulta en una matriz $n \times n$, llamada matriz de atracción de atributos.

La desventaja de este trabajo es que sólo se basa en la frecuencia de los atributos y en los atributos utilizados por las consultas, y no toma en cuenta el tamaño de los atributos para obtener los fragmentos.

Marir F. et al. [71] presenta un algoritmo para fragmentar verticalmente bases de datos relacionales que consiste en realizar un agrupamiento utilizando la AAM, este trabajo tiene dos factores para evitar obtener agrupamientos pobres entre atributos y entre grupos. Consideran que la principal ventaja del algoritmo es que es fácil de entender y fácil de implementar, ya que sólo tiene dos pasos: en el primero se obtienen grupos iniciales con base en los valores de afinidad entre los atributos, en el segundo paso se busca fusionar los grupos iniciales para producir los fragmentos finales. La desventaja de este algoritmo es que no toman en cuenta los tamaños de los atributos.

Abuelyaman E.S. [72] propone realizar una fragmentación vertical antes de conocer las frecuencias de acceso de las consultas a los atributos, su método de fragmentación se basa en una matriz de uso de atributos a la que ellos le llaman la matriz de reflexividad, utilizando esta matriz se construye una matriz de simetría que almacena el número de consultas que acceden dos atributos juntos (afinidad). Después se realiza una partición de esta matriz iniciando del atributo con menor accesos de consultas (menor reflexividad) y buscando posteriormente el atributo con mayor afinidad, esto se realiza hasta que se forma un ciclo o hasta que ya no hay más atributos por agrupar. Por último, se obtiene una tabla de aciertos y equivocaciones que contiene el número de veces que un atributo se relaciona con un miembro de su grupo y el número de veces que se relaciona con un miembro de otro grupo, si la proporción de aciertos es menor que un umbral determinado,

el atributo debe moverse a otro grupo para mejorar las tasas de acierto. Las desventajas de este trabajo es que la fragmentación se realiza con base en probabilidades, si no se tiene información de las consultas es muy difícil realizar una fragmentación adecuada, además no se dan lineamientos precisos para elegir el umbral ni para saber como mover el atributo con menores aciertos.

Propuestas basadas en costo Entre las propuestas que se basan en el costo de las consultas para realizar la fragmentación vertical tenemos que [45], [75], [76], [65] consideran que el tiempo de respuesta de una consulta esta fuertemente afectada por la cantidad de datos accedida de memoria secundaria (disco). Por lo tanto, su función objetivo es minimizar el número de accesos a disco, en estas propuestas la fragmentación vertical se realiza con base en un análisis de consultas para minimizar el número de accesos a disco.

Hammer y Niamir [45] asignan cada atributo a un fragmento distinto, y en cada paso, se consideran todos los posibles agrupamientos de esas particiones y se elige el agrupamiento que tenga el menor costo como el agrupamiento candidato para la siguiente iteración, posteriormente, se intenta mover atributos en los fragmentos, esto se realiza hasta que no se obtiene ningún beneficio. Esta propuesta se considera ascendente y se basa en el uso de un evaluador de particiones que realiza un análisis de costo del procesamiento de transacciones para cada fragmento.

Cornell y Yu [75] realizan una fragmentación vertical basada en una técnica de programación entera que consiste de un paso de análisis y un paso de fragmentación.

Chu y Ieong [76] proponen un algoritmo de fragmentación binaria basado en transacciones con una función objetivo para minimizar el número total de accesos a disco.

Agrawal S. et al. [65] integran fragmentación vertical con fragmentación horizontal, vistas e índices para obtener un diseño físico de bases de datos con mejor desempeño y manejabilidad. En esta propuesta no restringen su espacio a fragmentación binaria debido a que realizan una fusión de los fragmentos verticales.

Las desventajas de estas propuestas es que están limitados a un ambiente de un solo

sitio y no pueden extenderse fácilmente a sistemas distribuidos, ya que el costo de ejecutar consultas en sitios distribuidos esta dominada por la comunicación remota en la red además de los accesos locales a disco.

Debido a la necesidad de una función objetivo común que permitiera evaluar distintos algoritmos de fragmentación vertical, Chakravarthy et al. [73] presentan un evaluador de particiones (PE, en inglés: Partition Evaluator) para algoritmos que utilicen la matriz de uso de atributos como entrada, el cual tiene dos componentes: el primero mide el costo de procesamiento local de transacciones debido al acceso a atributos locales irrelevantes, el segundo mide el costo de procesamiento remoto debido al acceso de transacciones a atributos remotos relevantes. El objetivo del PE es encontrar un esquema de fragmentación vertical que minimice el costo del procesamiento de transacciones en las bases de datos distribuidas. Sin embargo, el costo de acceso a atributos remotos relevantes refleja el número de atributos relevantes en un fragmento accedidos remotamente con respecto a todos los otros fragmentos por todas las transacciones. Por lo tanto, el PE no puede usarse en bases de datos distribuidas porque no toma en cuenta los factores de costo de las transacciones de red, ni el tamaño de los datos.

La mayoría de las propuestas previas se enfocaban en generar una fragmentación óptima sin considerar el número de fragmentos (fragmentación de mejor ajuste), sólo algunos trabajos toman en cuenta los casos en los que es necesario generar cierto número de fragmentos por la fragmentación vertical [36], [75], [65] (fragmentación vertical n-aria). Son y Kim [74] consideran que puede ser más efectivo y útil un método que soporte ambos tipos de fragmentación vertical, por lo tanto, presentan un método de fragmentación vertical adaptable con el que es posible realizar fragmentación vertical n-aria, así como fragmentación vertical de mejor ajuste que se basa en un modelo de costo con el que buscan minimizar no sólo la frecuencia de acceso de consultas a diferentes fragmentos sino también la frecuencia de los accesos interferidos entre las consultas. Las desventajas de esta propuesta es que la función objetivo no toma en cuenta el costo de acceso a atributos irrelevantes (reducción de accesos a disco) ni el tamaño de los atributos.

Propuestas basadas en minería de reglas de asociación La minería de datos es un proceso para extraer información importante, implícita, previamente desconocida y potencialmente útil de los datos almacenados en una base de datos [80]. Uno de los problemas más importante en minería de datos es el descubrimiento de reglas de asociación para bases de datos grandes, a esto se le llama minería de reglas de asociación.

El propósito de la minería de reglas de asociación es el descubrimiento de la ocurrencia mutua de asociaciones entre los datos en bases de datos grandes, esto es, encontrar elementos que implican la presencia de otros elementos en la misma transacción [81].

Definición 3.10 *Dado un multiconjunto finito de transacciones D, el problema de minería de reglas de asociación es generar todas las reglas de asociación que tengan, al menos, el mismo soporte y la misma confianza que los umbrales especificados por el usuario de soporte mínimo (min-sup) y confianza mínima (min-conf), respectivamente [82].*

Reglas de asociación

Definición 3.11 *Dado un conjunto de transacciones D, donde cada transacción contiene un conjunto de elementos, una regla de asociación se define como una expresión* $X \rightarrow Y$*, donde X y Y son conjuntos de elementos y* $X \cap Y = \oslash$*. La regla implica que las transacciones de la base de datos que contienen X tienden a contener Y [81].*

Soporte y confianza son dos medidas de asociación. El factor soporte indica la ocurrencia relativa de X y Y en el conjunto total de transacciones y se define como la proporción del número de tuplas que satisfacen X y Y sobre el número total de tuplas. El factor confianza es la probabilidad de Y dada X y se define como la proporción del número de tuplas que satisfacen X y Y sobre el número de tuplas que satisfacen X. El factor soporte indica la frecuencia de la ocurrencia de patrones y el factor confianza denota la fuerza de implicación de la regla [80].

Suponiendo que N es el número total de tuplas y $|A|$ es el número de tuplas que contienen todos los elementos en el conjunto A. Definimos

$$support(X) = p(X) = \frac{|X|}{N} \tag{3.1}$$

$$support(X \rightarrow Y) = p(X \cup Y) = \frac{|X \cup Y|}{N} \tag{3.2}$$

$$confidence(X \rightarrow Y) = \frac{P(X \cup Y)}{P(X)} = \frac{|X \cup Y|}{|X|} \tag{3.3}$$

El problema es encontrar todas las reglas de asociación que satisfagan las restricciones de soporte y confianza mínimos especificadas por el usuario que se tienen en una base de datos dada [81].

Fragmentación vertical basada en reglas de asociación Gorla and Betty [77] desarrollaron un algoritmo de fragmentación vertical basado en reglas de asociación. Este consiste en cuatro pasos: 1) primero, tiene que descubrir conjuntos grandes de elementos, los cuales son combinaciones de atributos que tienen soporte mayor al soporte mínimo predefinido (*min-sup*). El algoritmo Apriori [83] se adapta para este paso, 2) el algoritmo filtra los conjuntos grandes de elementos, esto es, descarta los conjuntos grandes de elementos con un valor de confianza menor al umbral de confianza mínima especificado por el usuario *(min-conf)*, 3) deriva las particiones seleccionado los conjuntos grandes de elementos disjuntos. En este paso el algoritmo genera varios esquemas de fragmentación vertical (EFVs) y, finalmente, 4) los EFVs se evalúan para encontrar el EFV óptimo.

La principal desventaja de este algoritmo es que encuentra distintos EFVs utilizando diferentes valores *min-sup* y *min-conf*. Encontrar el esquema óptimo implica determinar el valor apropiado de dichos umbrales, pero no se da ninguna guía para hallar los valores correctos. Incluso con los valores adecuados de *min-sup* y *min-conf*, el algoritmo encontrará varios EFVs y es necesario evaluarlos usando una función de costo con el fin de obtener el EFV óptimo. Sería más eficiente que el algoritmo pudiera fijar automáticamente los valores de estos umbrales, esto eliminaría la complejidad de determinar el valor adecuado de dichos

umbrales y el EFV óptimo.

Por estas razones creamos un método de fragmentación vertical, llamado SVP (En inglés: Support-Based Vertical Partitioning) que toma como entrada una AUM y también necesita un umbral de soporte mínimo (*min-sup*), pero este es determinado automáticamente por el algoritmo. SVP tiene tres pasos: en el primer paso obtiene una matriz de soporte de atributos (ASM, en inglés: Attribute Support Matrix) de la AUM; en el segundo paso determina el valor de *min-sup* y, finalmente, encuentra los fragmentos óptimos usando el algoritmo de fragmentación basado en conexiones (CBPA, en inglés: Connection Based Partitioning Algorithm) [79]. A continuación se describe nuestra propuesta.

Propuesta de fragmentación vertical basada en reglas de asociación

Los algoritmos de fragmentación vertical requieren como entrada una AUM. La Tabla 3.1 muestra una AUM de una relación con seis atributos y cuatro consultas. [74]. Las entradas 0/1 en la AUM muestran si las consultas (Q) usan los atributos (A) o no, y la columna de acceso (F) muestra la frecuencia de acceso de las consultas a los atributos por un periodo de tiempo (por ejemplo un día).

Debido a que la fragmentación vertical coloca en un fragmento aquellos atributos que las consultas acceden juntos usualmente, es necesaria una medida que defina más precisamente la noción de "unión", esta medida es la afinidad de atributos que indica qué tan relacionados son los atributos [11].

Tabla 3.1: Matriz de uso de atributos (AUM).

Q/A	a_1	a_2	a_3	a_4	a_5	a_6	F
q_1	0	1	1	1	0	0	$f_1 = 15$
q_2	1	1	0	0	1	1	$f_2 = 10$
q_3	0	0	0	1	1	1	$f_3 = 25$
q_4	0	1	1	0	0	0	$f_4 = 20$

La principal desventaja de la afinidad es que no muestra la "unión" cuando se involucran más de dos atributos. Por lo tanto, esta medida no tiene relación con la afinidad medida

con respecto a los grupos de atributos completos [73]. Para resolver este problema se define un factor de soporte basado en afinidad para fragmentación vertical como sigue:

$$support(a_i, a_j) = \frac{aff(a_i, a_j)}{\sum_{k=1}^{m} f_k} \tag{3.4}$$

Aquí $support(a_i,a_j)$ muestra la probabilidad de acceder a ambos atributos i y j con respecto a todos los accesos de las consultas. Estos valores se usan en un algoritmo de fragmentación vertical basado en soporte para encontrar el EFV óptimo.

Algoritmo de Fragmentación Vertical basado en Soporte (SVP) El Algoritmo 3.1 muestra el procedimiento de SVP, este tiene tres pasos: generar la ASM, determinar el valor del umbral y obtener el EFV óptimo.

Algoritmo 3.1 SVP
Entrada:
Frecuencia de acceso total de las consultas (*total_ acc*)
AUM: Matriz de Uso de Atributos
Salida:
Esquema de Fragmentación Vertical óptimo (VPS)
begin
{Paso 1: Generar la ASM}
getASM(*total_ acc*, AUM, ASM)
{generar la ASM de la AUM usando la ecuación 3.4}
{Step 2: Determinar el valor del umbral}
getSVM(ASM,SVM)
{obtener la SVM de la ASM}
quick_sort(SVM)
{ordenar la SVM}
setThreshold(SVM, *min-sup*}
{encontrar el valor apropiado de *min-sup*}
{Step 3: Obtener el EFV óptimo}
CBPA(ASM, *min-sup*, EFV)
{obtener el EFV óptimo usando el valor de *min-sup* encontrado}
end. {SVP}

Generar la ASM En el primer paso, se obtiene la ASM usando la AUM de la relación que se va a fragmentar y la variable *total_ acc* que es la suma de la frecuencia de acceso

de todas las consultas. Esto se presenta en el Algoritmo 3.2. ASM es una matriz simétrica $n \times n$ (donde n es el número de atributos). Cada elemento de ASM es una de las medidas de soporte definidas en la ecuación 3.4.

Algoritmo 3.2 getASM

Entrada: AUM: Matriz de Uso de Atributos, *total_ acc.*
Salida: ASM: Matriz de Soporte de Atributos

```
begin
  support ← 0
  for i from 1 to n by 1 do
    for k  from 1 to q by 1 do
      if AUM_{k,i} = 1 then
        support ← f_k/total_ acc
        ASM_{i,j} ← ASM_{i,j}+support
        for j from i+1 to n by 1 do
          if AUM_{k,i} = 1 & AUM_{k,j} = 1 then
            support ← f_k/total_ acc
            ASM_{i,j} ← ASM_{i,j}+support
            ASM_{j,i} ← ASM_{i,j}
          end_if
        end_for
      end_if
    end_for
  end_for
end. {getASM}
```

La Tabla 3.2 muestra la ASM de la AUM presentada en la Tabla 3.1. Cada valor nos da el porcentaje de las consultas que contienen dos atributos juntos y la diagonal nos da el porcentaje de las consultas que contienen un atributo. Por ejemplo, el 57 % de las consultas usan el atributo a_4 ($ASM_{4,4}$), en el 36 % de las consultas el atributo a_4 se usa con los atributos a_5 y a_6 ($ASM_{4,5}$, $ASM_{4,6}$) y en el 21 % con los atributos a_2 y a_3 ($ASM_{4,2}$, $ASM_{4,3}$).

Determinar el valor del umbral Usamos el algoritmo de fragmentación basado en conexiones (CBPA) [79] para obtener los fragmentos. Este algoritmo encuentra fragmentos tratando de conectar atributos. Dos atributos pertenecen al mismo fragmento sólo si están "conectados", esto es, si el valor de soporte de dichos atributos es mayor o igual a *min-sup*.

Tabla 3.2: Matriz de soporte de atributos (ASM) de la AUM.

	a_1	a_2	a_3	a_4	a_5	a_6
a_1	0.14	0.14	0	0	0.14	0.14
a_2	0.14	0.64	0.5	0.21	0.14	0.14
a_3	0	0.5	0.5	0.21	0	0
a_4	0	0.21	0.21	0.57	0.36	0.36
a_5	0.14	0.14	0	0.36	0.5	0.5
a_6	0.14	0.14	0	0.36	0.5	0.5

Por lo tanto, es necesario encontrar los valores adecuados de *min-sup* para obtener el EFV óptimo.

CBPA no encuentra la solución óptima si la mayoría (al menos la mitad +1) de los atributos pueden clasificarse como atributos conectados, lo que significa que el valor de *min-sup* debe elegirse cuidadosamente. Por lo tanto, determinamos que el valor adecuado de *min-sup* debe ser el valor de soporte para el 71 % al 75 % de los datos. Para determinar el valor del umbral, SVP utiliza el Algoritmo 3.3 que genera una Matriz de Valores de Soporte (SVM, en inglés: Support Value Matrix) de la ASM. Este algoritmo obtiene una matriz $2 \times number$ (donde, *number* es el número de diferentes valores de soporte de una ASM), la cual almacena los diferentes valores de soporte y sus ocurrencias.

En el ejemplo, getSVM (Algoritmo 3.3) toma como entrada la ASM de la Tabla 3.2 y genera la SVM presentada en la Tabla 3.3. Como puede verse, en este caso *number*=7 porque se tienen siete diferentes valores de soporte en la ASM, así que getSVM produce una matriz 2×7, la primera fila de la matriz (Valor de Soporte) presenta los diferentes valores de soporte y la segunda fila (Ocurrencia) almacena el número de veces que cada valor aparece en la ASM.

Tabla 3.3: Matriz de valores de soporte (SVM) de la ASM.

	1	2	3	4	5	6	7
Valor de Soporte, SV	0.14	0	0.64	0.5	0.21	0.57	0.36
Ocurrencia, OC	11	8	1	7	4	1	4

Algoritmo 3.3 getSVM

input: ASM: Matriz de Soporte de Atributos
output: SVM: Matriz de Valores de Soporte
begin
number ← 0
ban ← 0
for i **from** 1 **to** n **by** 1 **do**
for j **from** 1 **to** n **by** 1 **do**
for s **from** 1 **to** number **by** 1 **do**
if $\text{ASM}_{i,j} = \text{SVM}_{SV,s}$ **then**
$\text{SVM}_{OC,s} \leftarrow \text{SVM}_{OC,s} + 1$
ban ← 1
s ← number
end-if
end_for
if ban=0 **then**
number ← number+1
$\text{SVM}_{SV,number} \leftarrow \text{ASM}_{i,j}$
$\text{SVM}_{OC,number} \leftarrow 1$
end_if
ban ← 0;
end_for
end_for
end. {getSVM}

Tabla 3.4: SVM ordenada

	1	2	3	4	5	6	7
Valor de Soporte, SV	0	0.14	0.21	0.36	0.5	0.57	0.64
Ocurrencia, OC	8	11	4	4	7	1	1

Tabla 3.5: SVM completa

	1	2	3	4	5	6	7
Valor de Soporte, SV	0	0.14	0.21	0.36	0.5	0.57	0.64
Ocurrencia, OC	8	11	4	4	7	1	1
Acumulación, AC	8	19	23	27	34	35	36

El siguiente paso del método propuesto es ordenar la SVM en orden creciente usando el algoritmo quick sort, usamos este algoritmo porque es uno de los algoritmos de ordenamiento más rápidos y simples, este tiene un tiempo de ejecución promedio de $\theta(n\ log\ n)$ [84]. La Tabla 3.4 muestra la SVM después de aplicar quick sort.

Para encontrar el valor apropiado de *min-sup* usamos el Algoritmo 3.4. Primero, las ocurrencias de cada valor (segunda fila de la SVM) se acumulan en la tercera fila de la SVM (Acumulación). Esto se muestra en la Tabla 3.5.

Después, el número total de datos (n^2, donde n es el número de atributos de la relación, en el ejemplo n=6, así que el número total de datos es 36) se multiplica por 0.75 y 0.71 respectivamente, esto se realiza porque el valor adecuado de *min-sup* debe ser el valor de soporte que se encuentra entre el 71 % y el 75 % de los datos, un valor menor que el 71 % clasificará como atributos conectados a la mayoría de los atributos, mientras que con un valor mayor al 75 % será muy difícil que un atributo se clasifique como conectado, estos extremos evitarán que se obtenga el EFV óptimo. Regresando al ejemplo, el resultado de la multiplicación es 27 y 25.56. Entonces, se busca en la tercera fila de SVM (Acumulación) un valor de acumulación mayor o igual al primer valor obtenido (27 en el ejemplo), cuando se encuentra dicho valor ($SVM_{AC,4}$), el valor previo se evalúa ($SVM_{AC,3}$), si es igual o mayor al segundo valor obtenido (25.56 en el ejemplo), entonces *min-sup* es igual al SV

Algoritmo 3.4 setThreshold

Entrada: SVM: Matriz de Valores de Soporte
Salida: *min-sup*
begin
  accum ← 0
  for *i* **from** 1 **to** *number* **by** 1 **do**
    accum ← *accum* + $\text{SVM}_{2,i}$
    $\text{SVM}_{3,i}$ ← *accum*
  end_for
  total_number ← $\text{SVM}_{3,number}$
  percentage ← *total_number* * 0.75
  percentage2 ← *total_number* * 0.71
  for *i* **from** 1 **to** *number* **by** 1 **do**
    if $\text{SVM}_{3,i} \geq$ *percentage* **then**
      if $\text{SVM}_{3,i-1} \geq$ *percentage2* **then**
        min-sup ← $\text{SVM}_{1,i-1}$
        i ← *number*
      else
        min-sup ← $\text{SVM}_{1,i}$
        i ← *number*
      end_if
    end_if
  end_for
end. {setThreshold}

de la segunda columna comparada ($SVM_{SV,3}$), de otra forma *min-sup* es igual al SV de la primera columna comparada ($SVM_{SV,4}$). En la Tabla 3.5 se puede ver que $SVM_{AC,4} = 27$, pero $SVM_{AC,3} \ngeq 25.56$, así que *min-sup*=0.36.

Obtener el EFV óptimo Finalmente, para obtener el EFV óptimo se utiliza el algoritmo CBPA (Algoritmo 3.5), la entrada a este algoritmo es la ASM y el umbral *min-sup* determinado en el paso previo. En el ejemplo el EFV obtenido con *min-sup*=0.36 es (a_1), (a_2,a_3), (a_4,a_5,a_6) y este es el EFV óptimo generado por Son y Kim [74] usando su algoritmo de fragmentación vertical adaptable (AVP, en inglés: Adaptable Vertical Partitioning).

Algoritmo 3.5 CBPA

Entrada: ASM: Matriz de Soporte de Atributos, *min-sup*
Salida: VPS: Esquema de Fragmentación Vertical
begin
{inicializa un arreglo VPS de n elementos con 0}
$f \leftarrow 1$
for i **from** 1 **to** n **by** 1 **do**
  if VPS_i=0 **then**
    $VPS_i \leftarrow f$
    for j **from** i **to** n **by** 1 **do**
      if $ASM_{i,j} \geq$ *min-sup* and VPS_j=0 **then** $VPS_j \leftarrow f$
    end_for
    $f \leftarrow f + 1$
  end-if
end-for
end.{CBPA}

Experimentos En esta sección se muestran varios resultados experimentales para evaluar la eficiencia del algoritmo SVP.

Para el primer experimento, se consideró la AUM de la Tabla 3.6, que se usó como un ejemplo por Gorla y Betty [77]. La Tabla 3.7 muestra la SVM. En este caso, el valor del umbral encontrado usando el algoritmo setThreshold fue *min-sup*=0.6 porque $SVM\text{-}2_{AC,3}$>36*0.75=27 y $SVM\text{-}2_{AC,2} \ngeq$ 36*0.71=25.56, el EFV óptimo generado fue (a_1,a_4,a_6), (a_2,a_5), (a_3) y este es la solución óptima encontrada por Gorla y Betty.

Tabla 3.6: Matriz de uso de atributos 2 (AUM-2).

queries/attributes	a_1	a_2	a_3	a_4	a_5	a_6	F
q_1	1	1	0	0	1	0	$f_1 = 1$
q_2	0	1	0	0	1	0	$f_2 = 3$
q_3	1	0	0	1	0	1	$f_3 = 3$
q_4	1	1	1	1	1	1	$f_4 = 2$
q_5	1	1	1	1	1	1	$f_5 = 1$

Tabla 3.7: SVM-2 completa.

	1	2	3	4
Valor de Soporte, SV	0.3	0.4	0.6	0.7
Ocurrencia, OC	19	4	8	5
Acumulación, AC	19	23	31	36

Se usó el ejemplo de la Tabla 3.8 como segundo experimento, el cual se presentó en [36]. La Tabla 3.9 muestra la SVM de este ejemplo. El valor del umbral determinado por el algoritmo *setThreshold* en este experimento fue *min-sup*=0.18 porque SVM-$3_{AC,5}$=100*0.75=75 y SVM-$3_{AC,4}$=100*0.71=71, se encontró el mismo EFV óptimo que en [36] (a_1,a_5,a_7), (a_2,a_3,a_8,a_9), (a_4,a_6,a_{10}).

En el tercer experimento, usamos la AUM también utilizada en [36] presentada en las Tablas 3.10 y 3.11. La Tabla 3.12 muestra la SVM. En este ejemplo, el valor encontrado

Tabla 3.8: AUM-3.

Q/A	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	Acc
q_1	1	0	0	0	1	0	1	0	0	0	25
q_2	0	1	1	0	0	0	0	1	1	0	20
q_3	0	0	0	1	0	1	0	0	0	1	25
q_4	0	1	0	0	0	0	1	1	0	0	35
q_5	1	1	1	0	1	0	1	1	1	0	25
q_6	1	0	0	0	1	0	0	0	0	0	25
q_7	0	0	1	0	0	0	0	0	1	0	25
q_8	0	0	1	1	0	1	0	0	1	1	15

Tabla 3.9: SVM-3 completa.

	1	2	3	4	5	6	7	8	9	10
Valor de Soporte, SV	0	0.07	0.11	0.18	0.22	0.27	0.33	0.38	0.49	0.51
Ocurrencia, OC	30	12	20	9	4	4	12	1	4	4
Acumulación, AC	30	42	62	71	75	79	91	92	96	100

Tabla 3.10: AUM-4 parte I.

Q/A	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	F
q_1	1	0	0	1	1	1	0	1	0	0	50
q_2	0	1	0	0	0	0	0	0	1	0	50
q_3	0	0	1	0	0	0	1	0	0	1	50
q_4	0	0	0	0	0	0	0	0	0	0	50
q_5	1	0	0	0	1	0	0	1	0	0	15
q_6	1	0	0	0	0	0	0	1	0	1	15
q_7	0	0	1	0	0	0	1	0	0	1	15
q_8	0	1	0	0	0	0	0	0	0	0	15
q_9	0	1	0	0	1	0	0	0	0	0	10
q_{10}	1	0	0	0	0	0	0	0	1	0	10
q_{11}	1	1	1	0	1	1	0	0	1	0	10
q_{12}	0	0	0	1	0	0	1	0	0	1	10
q_{13}	0	0	0	0	0	0	0	1	0	0	10
q_{14}	1	1	1	1	1	1	1	1	1	0	5
q_{15}	0	0	0	0	0	0	0	0	0	0	5

por el algoritmo *setThreshold* fue *min-sup*=0.12 porque SVM-$4_{AC,9}$>400*0.75=300 y SVM-$4_{AC,8} \geq$ 400*0.71=284, el EFV óptimo generado es (a_1, a_4, a_5, a_6, a_8), (a_2, a_9, a_{12}, a_{13}, a_{14}), (a_3, a_7, a_{10}, a_{11}, a_{17}, a_{18}), (a_{15}, a_{16}, a_{19}, a_{20}) y esta es la solución óptima encontrada por [36].

Como se demuestra con los experimentos, SVP siempre encontró el EFV óptimo. SVP resuelve los problemas del algoritmo de Betty y Gorla porque sólo genera una solución, la cual es la óptima. El algoritmo de Betty y Gorla obtiene diferentes soluciones y es necesario usar una función de costo para evaluarlas y encontrar la óptima. Además SVP resuelve el problema de determinar el valor de los umbrales que tienen los algoritmos de Betty y Gorla y CBPA.

Tabla 3.11: AUM-4 parte II.

Q/A	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}	a_{20}	Acc
q_1	0	0	0	0	0	0	0	0	0	0	50
q_2	0	1	1	1	0	0	0	0	0	0	50
q_3	1	0	0	0	0	0	1	1	0	0	50
q_4	0	0	0	0	1	1	0	0	1	1	50
q_5	0	0	0	0	0	0	0	0	0	0	15
q_6	1	0	0	0	0	0	0	0	0	0	15
q_7	1	0	0	0	0	0	1	0	0	0	15
q_8	0	1	1	1	1	1	0	1	0	1	15
q_9	1	0	0	1	0	0	0	0	1	0	10
q_{10}	0	0	0	0	0	1	0	1	0	0	10
q_{11}	0	1	1	0	0	0	0	0	0	0	10
q_{12}	0	0	0	1	0	0	0	0	1	1	10
q_{13}	1	0	0	0	1	1	1	1	0	0	10
q_{14}	0	0	0	0	0	0	0	0	0	0	5
q_{15}	0	0	0	0	1	1	1	1	1	1	5

Tabla 3.12: SVM-4 completa.

	1	2	3	4	5	6	7	8	9	10
SV	0	0.02	0.03	0.05	0.06	0.08	0.09	0.12	0.16	0.17
OC	122	30	70	60	4	8	2	2	8	14
AC	122	152	222	282	286	294	296	298	306	320
	11	12	13	14	15	16	17	18	19	20
SV	0.19	0.20	0.22	0.23	0.25	0.27	0.29	0.30	0.31	0.33
OC	6	28	8	16	11	3	5	1	1	1
AC	326	354	362	378	389	392	397	398	399	400

Fragmentación vertical en bases de datos orientadas a objetos

La mayoría de los algoritmos desarrollados en las bases de datos orientadas a objetos (BDOOs) se basan en los trabajos realizados sobre fragmentación vertical en bases de datos relacionales, los algoritmos que más han influido en las BDOOs son el algoritmo de fragmentación vertical binaria de Navathe [36], el algoritmo gráfico de Navathe y Ra [36] y el evaluador de particiones de Chakravarthy [73]. En las bases de datos orientadas a objetos

existen trabajos basados en afinidad [85], [86], [87], [88], costo [20], [89], [21], [90], [21] y conocimiento [91], [92].

Propuestas basadas en afinidad Ezeife y Barker realizan una clasificación de los métodos y los atributos de una clase en simples y complejos y desarrolla algoritmos para clases que consisten de atributos y métodos simples [85], atributos simples/complejos y métodos complejos/simples [87] y atributos y métodos complejos [86], utilizan el algoritmo de Navathe et al. [36] para realizar la fragmentación vertical de los métodos de la clase con base en la afinidad entre métodos, la diferencia es que toman en cuenta el enlace de herencia, composición de clases y métodos complejos incorporando los accesos de las subclases, clases contenedoras (que usan esta clase como atributo) y clases de métodos complejos a los atributos de la clase. Para cada modelo, se calculan diferentes matrices de afinidad que incorporan todas las características orientadas a objetos. La complejidad de dichos algoritmos es $O(cqm+m^2+fma)$ donde f es el número máximo de fragmentos en una clase, m es el número máximo de métodos en una clase, a es el número máximo de atributos en una clase, c es el número de clases en la base de datos y q es el número de aplicaciones (consultas) que acceden una base de datos, por lo tanto, es un algoritmo de tiempo polinómico. Todos los algoritmos presentados tienen como objetivo dividir una clase de tal forma que se agrupen juntos los atributos y métodos accedidos con mayor frecuencia, la desventaja es que no toman en cuenta los costos de transporte de datos, ni la ubicación de fragmentos. Además, estos algoritmos se compararan usando el PE de Chakravarthy et al. [73] que no mide realmente los costos de consulta totales.

Bellatreche et al. [88] presentan un algoritmo de fragmentación vertical para un modelo de atributos y métodos complejos, que al igual que Ezeife y Barker [86], utiliza la afinidad de métodos para formar los fragmentos de clases verticales, la diferencia es que utiliza el algoritmo gráfico de Navathe y Ra [37] para realizar la fragmentación vertical y que toman en cuenta las consultas realizadas por las superclases que acceden a atributos de la clase que se va a fragmentar y no de las subclases, clases contenidas y clases de métodos complejos.

Las desventajas de este algoritmo son las mismas que las del algoritmo gráfico [37]: que no toman en cuenta la ubicación de los fragmentos, el tamaño de los datos y los costos de transporte de datos a través de la red.

Propuestas basadas en costo Malinowski y Chakravarthy [89] presentan un evaluador de particiones orientado a objetos (PEOO, en inglés: Partition Evaluator for Distributed Object-Oriented Database), que es una extensión del PE relacional [73], para esto utilizan las matrices de uso de transacciones y métodos (TMUM, Transaction-Method Usage Matrix), métodos y métodos (MMUM, Method-Method Usage Matrix), y métodos y atributos (MAUM, Method-Attribute Usage Matrix) y las multiplican para obtener la matriz de uso de transacciones y atributos (TAUM, Transaction Attribute Usage Matrix) que es igual a la matriz de uso de atributos del modelo relacional. Posteriormente realizan una búsqueda exhaustiva utilizando el PEOO y la TAUM para encontrar el mejor esquema de fragmentación. Muestran el uso del PEOO en todos los modelos de clases (atributos simples/complejos y métodos simples/complejos). La función objetivo propuesta trata de balancear el costo de accesos locales y remotos para transacciones dadas. La principal diferencia entre el PEOO y el PE es que el primero considera que una misma transacción puede tener diferentes frecuencias de acceso a los atributos, mientras que para el segundo la frecuencia de una transacción siempre es la misma. El PEOO no mide los costos totales de las consultas, ya que sólo considera el acceso a atributos remotos relevantes y a atributos locales irrelevantes y no el tamaño de los datos transferidos entre los nodos de la red.

Fung et. al [90] presentan un modelo de costo para estudiar la efectividad de la fragmentación vertical en BDOOs en términos de reducir el número de accesos a disco para procesar una consulta. Presentan tres algoritmos para encontrar EFVs óptimos o casi óptimos. El primer algoritmo usa el modelo de costo para encontrar el EFV óptimo. El segundo algoritmo emplea el algoritmo gráfico de Navathe y Ra [37], por lo tanto, se basa en afinidad. El último algoritmo es una propuesta mixta que primero aplica el algoritmo basado en afinidad para obtener un EFV inicial, y entonces aplica una heurística de ascenso

de colinas (hill-climbing) para mejorar la solución (si es posible) usando el modelo de costo.

En [21] comparan las propuestas de fragmentación vertical basada en costo y basada en afinidad. La propuesta basada en afinidad no modela el tamaño de los objetos/atributos y este es un factor muy importante en el costo del procesamiento de consultas. Su propuesta basada en costo modela el tamaño de los atributos/objetos y muestran que la propuesta basada en costo se desempeña mejor que la propuesta basada en afinidad.

Posteriormente en [20] desarrollan un modelo de costo más general para el procesamiento de consultas en clases de bases de datos orientadas a objetos fragmentadas verticalmente que incluye jerarquía de composición y jerarquía de subclases. Se concentran en la fragmentación no superpuesta y completa. Este modelo de costo no es adecuado para bases de datos distribuidas porque no toma en cuenta el costo de transferencia de los datos a través de la red, sino que sólo considera el número de accesos a disco.

Propuestas basadas en conocimiento Cruz F. [91] presenta una propuesta basada en conocimiento para el problema de la fragmentación vertical durante el diseño de bases de datos orientadas a objetos distribuidas. Extienden el algoritmo de Navathe y Ra [37] para aplicarlo en las bases de datos orientadas a objetos y lo implementan como un conjunto de reglas Prolog para usarlo como conocimiento de fondo en un proceso de revisión de teoría, con el objetivo de descubrir un nuevo algoritmo modificado, que proponga esquemas de fragmentación verticales óptimos (o casi óptimos) con desempeño mejorado. Utilizan un método de aprendizaje máquina llamado Inductive Logic Programming (ILP), que ejecuta un proceso de descubrimiento/revisión de conocimiento usando el conjunto de reglas como conocimiento de fondo. El objetivo de este trabajo es descubrir nuevas heurísticas que puedan considerarse en el proceso de fragmentación vertical. No toman en cuenta la ubicación de los fragmentos, el tamaño de los datos, el costo de transporte de datos a través de la red, no realizan una evaluación de su algoritmo para demostrar que los fragmentos obtenidos con su método mejoran el procesamiento de consultas.

Rajan John y V. Saravanan [92] proponen utilizar agentes inteligentes para automatizar

la fragmentación vertical de bases de datos orientados a objetos, básicamente los agentes se encargan de las tareas de identificación de clases, de objetos, de atributos y de métodos, pero no presentan ningún algoritmo, ni especifican como funciona su método, tampoco comparan su método con otros.

Ventajas y desventajas de la fragmentación vertical

Debido a que con frecuencia las consultas no requieren todos los atributos de una tupla (objeto), la fragmentación vertical almacena juntos aquellos atributos que las consultas requieren juntos con mayor frecuencia, por lo tanto, reduce el número de páginas que se transfieren de disco a memoria principal y los datos remotos transportados en el procesamiento de una consulta.

En aplicaciones de bases de datos como gestión de documentos, aplicaciones multimedia e hipermedia, muchos de los atributos tienden a ser objetos de gran tamaño, y no deben recuperarse si no son accedidos por las consultas. Las técnicas de fragmentación vertical pueden utilizarse en estas aplicaciones para proveer un rápido acceso a los datos [21].

La desventaja de la mayoría de los algoritmos, tanto en bases de datos relacionales como orientadas a objetos, es que no toman en cuenta el tamaño de los atributos [89], [37], [70], [62], [71], [72], [73], [74], [88], [91] y este es un factor muy importante en el procesamiento de consultas, especialmente en el caso de aplicaciones de bases de datos multimedia donde los atributos tienen tamaños muy variados, algunos algoritmos si consideran el tamaño de los datos [20], [90], [93], [75], pero no son adecuados para utilizarse en bases de datos distribuidas porque se basan en la idea de que el costo de consultas se ve afectado solo por el número de accesos a disco y descartan el costo de transferencia de los datos a través de la red, que es el factor que domina los costos de las consultas [85], [86], [87].

3.1.3. Fragmentación híbrida

La fragmentación híbrida o mixta consiste en aplicar simultáneamente la fragmentación horizontal y vertical en una relación. Por lo tanto, en este tipo de fragmentación se divide

una relación R en un conjunto de fragmentos fr_1, fr_2, ..., fr_n, cada uno de los cuales contiene un subconjunto de los atributos y de las tuplas de R, así como la clave primaria de R. Esto puede lograrse de dos formas: ejecutando la fragmentación horizontal y después la fragmentación vertical o viceversa. La necesidad de fragmentación híbrida en bases de datos distribuidas surge porque los usuarios de la base de datos generalmente acceden subconjuntos de datos que son fragmentos verticales y horizontales de las relaciones globales y debido a la necesidad de procesar consultas que accedan estos fragmentos de manera óptima [61].

Definición 3.12 *Un fragmento híbrido fr_i se define usando operaciones de selección y proyección de álgebra relacional. Un fragmento híbrido para una relación dada R puede definirse como sigue:*

$$fr_i = \sigma_{p_i}(\prod_{a_1,a_2,...,a_n}(R)) \text{ o } fr_i = \prod_{a_1,a_2,...,a_n}(\sigma_{p_i}(R))$$

donde p_i es un predicado basado en uno o más atributos de la relación R y a_1, a_2, ..., a_n son atributos de la relación R [49].

Por ejemplo, los fragmentos híbridos de la figura 3.4 pueden definirse de la siguiente forma:

$$EQUIPO_1 = \sigma_{NOMBRE='MOTOBOMBA'}(\prod_{ID,NOMBRE,MCA,MOD,CAB}(EQUIPO))$$

$$EQUIPO_2 = \sigma_{NOMBRE='MOTOBOMBA'}(\prod_{ID,PRECIO,CANT}(EQUIPO))$$

$$EQUIPO_3 = \sigma_{NOMBRE='PODADORA'}(\prod_{ID,NOMBRE,MCA,MOD,CAB}(EQUIPO))$$

$EQUIPO_1$

ID	NOMBRE	MCA	MOD	CAB
MB1	Motobomba	Super	MT2AMD	8
MB2	Motobomba	Gimexsa	GBG-20	5.5

$EQUIPO_2$

ID	PRECIO	CANT
MB1	5000	20
MB2	3500	15

$EQUIPO_3$

ID	NOMBRE	MCA	MOD	CAB
PO1	Podadora	Rally	TVS90	3.8
PO2	Podadora	Truper	P-2240	4

$EQUIPO_4$

ID	PRECIO	CANT
PO1	2800	12
PO2	3000	10

Figura 3.4: Ejemplo de fragmentación híbrida.

$$EQUIPO_4 = \sigma_{NOMBRE='PODADORA'}(\prod_{ID,PRECIO,CANT}(EQUIPO))$$

En este caso los fragmentos híbridos de la figura 3.4 se obtienen aplicando primero la fragmentación vertical y posteriormente la fragmentación horizontal a la relación EQUIPO. Si los fragmentos se obtienen aplicando primero la fragmentación horizontal y posteriormente la fragmentación vertical a la relación EQUIPO, entonces se definen de la siguiente manera:

$$EQUIPO_1 = \prod_{ID,NOMBRE,MCA,MOD,CAB}(\sigma_{NOMBRE='MOTOBOMBA'}(EQUIPO))$$

$$EQUIPO_2 = \prod_{ID,PRECIO,CANT}(\sigma_{NOMBRE='MOTOBOMBA'}(EQUIPO))$$

$$EQUIPO_3 = \prod_{ID,NOMBRE,MCA,MOD,CAB}(\sigma_{NOMBRE='PODADORA'}(EQUIPO))$$

$$EQUIPO_4 = \prod_{ID,PRECIO,CANT}(\sigma_{NOMBRE='PODADORA'}(EQUIPO))$$

Algoritmos de fragmentación híbrida

Navathe et al. [61] propusieron una metodología de fragmentación híbrida para el diseño inicial de bases de datos distribuidas. La entrada al proceso de fragmentación híbrida comprende una tabla de afinidad de predicados y una tabla de afinidad de atributos, primero se crean un conjunto de celdas que pueden superponerse entre ellas, posteriormente se fusionan algunas celdas de tal forma que pueda reducirse los accesos a disco totales de todas las consultas. Finalmente, se remueve la superposición entre cada par de fragmentos usando dos algoritmos para los casos de fragmentos superpuestos y contenidos. Sin embargo, los algoritmos de fusión se basan en un modelo de costo que sólo mide los accesos a disco y no considera el costo de comunicación en la red. Para las bases de datos distribuidas no sólo es importante reducir el número de accesos a disco sino también reducir el transporte de datos entre sitios.

Baiao y Mattoso [94] extienden el trabajo de Navathe et al. [61] para aplicarlo en BDOOs, su propuesta se divide en tres fases. Primero hay una fase de análisis para asistir a los administradores de la base de datos (DBAs) en la definición de la técnica de fragmentación más adecuada (horizontal, vertical o híbrida) para que se aplique en cada clase del esquema de la base de datos. La segunda fase presenta y aplica un algoritmo para ejecutar la fragmentación vertical en una clase. Finalmente, se presenta un algoritmo para ejecutar la fragmentación horizontal en la clase completa o en un fragmento vertical de una clase, que puede resultar en fragmentación híbrida. La fase de análisis se refinó posteriormente en [95], y en [96] se presenta una definición formal de la propuesta de fragmentación mixta.

Ng et al. [97] proponen un algoritmo génetico para el diseño de la fragmentación híbrida. Su algoritmo tiene como objetivo reducir el costo de procesamiento de las transacciones, el cual consideran se mide en términos del número de páginas accedidas para realizar la

transacción. Comparan su método con la numeración exhaustiva y la búsqueda aleatoria y obtienen mejores resultados.

Ventajas y desventajas de la fragmentación híbrida

El desempeño de las consultas puede incrementarse considerablemente usando este tipo de fragmentación debido a que los fragmentos contienen menos atributos, así como menos tuplas (objetos). Sin embargo, la reconstrucción de la tabla es más complicada debido a que es necesario unir todos los fragmentos verticales y los fragmentos horizontales. Otra desventaja es que no existen muchos trabajos de investigación relacionados con este tipo de fragmentación en la literatura.

3.1.4. Comparación entre los tipos de fragmentación

La fragmentación horizontal, vertical e híbrida son aspectos importantes del diseño físico de base de datos que tiene un impacto considerable en el desempeño y la facilidad de administración de bases de datos relacionales y bases de datos orientadas a objetos. Al igual que los índices y las vistas materializadas, los tres tipos de fragmentación pueden impactar el desempeño de una carga de trabajo (conjunto de consultas) reduciendo el costo de acceder y procesar datos.

La fragmentación horizontal se ha implementado ampliamente en los DBMSs actuales. Dos tipos comunes de fragmentación horizontal soportadas en la mayoría de los DBMSs son fragmentación *rango* y *hash*. En la fragmentación hash, el número de particiones para una fila dada se genera aplicando una función hash definida por el sistema en un conjunto específico de columnas del objeto. Un método de fragmentación hash se define por una tupla (C,n) donde C es un conjunto de tipos columna y n es el número de fragmentos. Por ejemplo, supongamos que R es una relación (a_1 *int,* a_2 *int,* a_3 *float,* a_4 *date*). El método de fragmentación hash H_1 =*({int, int}, 10}* divide R en 10 fragmentos aplicando la función hash a los valores de los atributos a_1, a_2 en todas las tuplas de R.

Un método de fragmentación rango se define por una tupla (c, V), donde c es un

tipo columna, y V es una secuencia ordenada de valores del dominio de c. Por ejemplo, el método de fragmentación rango R_1=*(date, <'01-01-11', '01-01-12'>)* cuando se aplica en el atributo a_4 de la relación R, divide R en tres fragmentos, uno por rango definido por la secuencia de fechas. El primer rango se define por todos los valores $\leq$ '01-01-11', el segundo rango son todos los valores $>$ '01-01-11' y $<$ '01-01-12' y el tercer rango contiene todos los valores $\geq$ '01-01-12'.

Sin embargo, en la mayoría de los DBMS actuales (DB2, MySQL, Oracle 11g, SQL server and PostgreSQL) no hay soporte DDL (Lenguaje de Definición de Datos) nativo para definir fragmentos verticales (ni híbridos) de una relación [65]. La fragmentación vertical es difícil de implementar debido a que se basa completamente en las consultas del usuario, por lo tanto, es necesario monitorizar consultas con el fin de saber cómo agrupar los atributos para lograr un buen tiempo de respuesta. Como la fragmentación híbrida es una combinación de la fragmentación horizontal y vertical, la dificultad de implementación de la fragmentación vertical implica que la implementación de la fragmentación híbrida sea complicada también.

Una forma efectiva de monitorizar consultas es usando reglas activas [25] debido a las siguientes razones:

1. Permiten realizar monitorización continua dentro del servidor de la base de datos.

2. Tienen la habilidad de realizar acciones automáticamente con base en la monitorización.

3. Debido a su simplicidad pueden implementarse sin costos adicionales de memoria y CPU.

La fragmentación vertical es adecuada para emplearse en aplicaciones multimedia donde los objetos tienden a ser muy grandes y muchas de las consultas no requieren acceso a los objetos completos [20]. Por esta razón en el siguiente Capítulo proponemos un algoritmo de fragmentación vertical para bases de datos multimedia llamado MAVP.

3.2. Fragmentación estática en bases de datos multimedia

Las mayoría de técnicas de fragmentación actuales sólo consideran datos alfanuméricos. En años recientes, las aplicaciones multimedia distribuidas y los datos multimedia se han vuelto disponibles a una tasa creciente; como resultado, las técnicas de fragmentación de datos se han adaptado en un contexto multimedia para lograr apropiadamente una alta utilización de recursos e incrementar la concurrencia y el paralelismo [19].

3.2.1. Características de las bases de datos multimedia

Debido a que cada vez más y más datos multimedia digitales en la forma de imagénes, video y audio se capturan y almacenan, surgió la necesidad de desarrollar técnicas eficientes de almacenamiento y recuperación de datos.

Una base de datos multimedia (MMDB, en inglés: MultiMedia DataBase) es una colección controlada de elementos de datos multimedia, como texto, imágenes, gráfico, video y audio. Un DBMS multimedia provee soporte para tipos de datos multimedia, además de facilitar la creación, almacenamiento, acceso, consulta y control de la base de datos multimedia.

Los diferentes tipos de datos involucrados en bases de datos multimedia podrían requerir métodos especiales para almacenamiento, acceso, indexación y recuperación. El DBMS multimedia debe acomodar estos requerimientos especiales suministrando abstracciones de alto nivel para manejar los diferentes tipos de datos, junto con una interfaz adecuada para su presentación.

Los tipos de datos encontrados en una base de datos multimedia típica incluyen [10]:

- Texto.
- Imágenes: color, blanco y negro, fotografías, mapas y pinturas.
- Objetos gráficos: dibujos ordinarios, bosquejos, ilustraciones, objetos 3D.
- Secuencias de animación: imágenes u objetos gráficos, generados independientemente.

- Video: también una secuencia de imágenes (llamadas cuadros), pero normalmente registran un evento del mundo real y los produce una grabadora de video.
- Audio: generados por medio de un dispositivo de grabación de audio.
- Multimedia compuesta: formada de una combinación de dos o más de los tipos de datos anteriores, como la mezcla de audio y video con una anotación textual.

Actualmente nos enfrentamos a una explosión de información multimedia. Por ejemplo, una gran cantidad de imágenes y video se crea y almacena en Internet. Muchas pinturas y fotografías en forma impresa se están convirtiendo a formato digital para su fácil procesamiento, distribución y preservación. Fotografías de noticiarios y periódicos también se están convirtiendo a formato digital para fácil mantenimiento y preservación. Un gran número de imágenes médicas se están capturando cada día y los satélites están produciendo muchas más. Esta tendencia va a continuar con el avance de las tecnologías de almacenamiento y digitales. Es ineficiente crear un repositorio para esta cantidad de información multimedia que crece cada vez más, ya que sería imposible usar completamente esta información multimedia a menos que esté organizada para lograr una rápida recuperación en demanda.

No sólo se debe almacenar una cantidad creciente de datos, sino también los tipos de datos y las características de los datos multimedia son diferentes de los datos alfanuméricos. Las principales características de los datos multimedia son [22]:

- Los datos multimedia, en especial el audio y el video, tienen un tamaño muy grande. Por ejemplo, una secuencia de video de 10 minutos de calidad razonable requiere casi 1.5 GB de almacenamiento sin compresión.
- El audio y el video tienen una dimensión temporal y deben reproducirse a una tasa fija para lograr el efecto deseado.
- Muchas aplicaciones multimedia requieren presentación simultánea de múltiples tipos de datos multimedia en una forma coordinada tanto espacial como temporal.

- El significado de los datos multimedia algunas veces es difuso y subjetivo. Por ejemplo, diferente gente puede interpretar la misma fotografía en formas completamente diferentes.

- Los datos multimedia son ricos en información. Muchos parámetros se requieren para recuperar su contenido adecuadamente.

A continuación se presentan los métodos de fragmentación en bases de datos multimedia.

3.2.2. Fragmentación horizontal

Saad et al. [16] propusieron una técnica de fragmentación horizontal para bases de datos multimedia. Los autores discuten la fragmentación horizontal primaria multimedia basada en características de bajo nivel de datos multimedia (por ejemplo, color, forma, textura, etc., representados como vectores de características complejas). Ellos enfatizan particularmente la importancia de la implicaciones de predicados multimedia en la optimización de los fragmentos multimedia.

Getahun et al. [18], [17] abordan el problema de identificar especificaciones semánticas entre predicados multimedia basados en texto y proponen integrar bases de conocimiento como una estructura para evaluar las relaciones semánticas entre los valores de los predicados y los operadores.

Las propuestas anteriores no consideran características relacionadas a los datos multimedia cuando calculan la implicación entre predicados multimedia. Además sólo consideran el tipo de fragmentación horizontal primaria. Chbeir y Laurent [19] resuelven estos problemas, ellos consideran tanto características multimedia como características semánticas y definen un ordenamiento de las consultas de interés. Con base en este ordenamiento, obtienen un conjunto de consultas frecuentes (FQ, en inglés: Frequently-asked Queries) cuyas respuestas constituyen un conjunto de datos mínimo que tiene que almacenarse para optimizar el cálculo de las respuestas a otras consultas comparables a alguna que se encuentre

en FQ.

3.2.3. Fragmentación vertical

Aunque la mayoría de técnicas recientemente propuestas para MMDBs se basan en fragmentación horizontal, la fragmentación vertical se ha reconocido como una técnica adecuada para mejorar el desempeño de las consultas en MMDBs en [93], [20], [21]. Fung et al. [15] aplican fragmentación vertical en una base de datos de videos de un sistema de aprendizaje electrónico con el objetivo de lograr la ejecución eficiente de las consultas. Sin embargo, los fragmentos se obtienen usando un modelo de costo basado en el ahorro en el número de accesos a disco y no toma en cuenta el costo de transporte que es muy importante en MMDBs distribuidas [98].

3.3. Fragmentación dinámica en bases de datos tradicionales

3.3.1. Fragmentación horizontal

McIver y King [99] presentan una arquitectura para realizar reagrupamiento de objetos complejos en línea que es adaptable a los cambios en los patrones de acceso. Su arquitectura involucra la descomposición de un método de agrupamiento en componentes de ejecución concurrente, cada uno de los cuales manejan una de las tareas involucradas en el reagrupamiento, estas tareas son recolección de estadísticas, análisis de grupos y reorganización de grupos. Sus resultados muestran que la tasa de error promedio al acceder a los objetos se reduce significativamente usando una combinación de reglas que desarrollan para decidir cuándo deben ejecutarse el analizador de grupos y las reorganizaciones.

Ezeife y Zheng [59] afirman que es necesaria una refragmentación de una base de datos orientada a objetos cuando la información del esquema y de los patrones de acceso de la base de datos experimentan suficientes cambios. Así que proveen una técnica para medir el desempeño de los fragmentos horizontales situados en sitios distribuidos. Dicha medida es una extensión del evaluador de particiones (PE) propuesto por Chakravarthy et al. [73].

DYFRAM [14] es una propuesta de fragmentación horizontal dinámica y asignación de los fragmentos con base en un análisis previo de las consultas. DYFRAM también toma en cuenta la replicación de los fragmentos. Su objetivo es maximizar el número de accesos locales comparado con el número de accesos remotos.

Liroz-Gistau et al. [100] proponen DynPart, un algoritmo de fragmentación dinámica para bases de datos que crecen continuamente. DynPart distribuye eficientemente nuevos datos insertados, con base en su afinidad con los fragmentos existentes en la aplicación.

3.3.2. Fragmentación vertical

L. Zhenjie [47] presenta una propuesta de fragmentación vertical dinámica para mejorar el desempeño de consultas en bases de datos relacionales, su propuesta se basa en el *ciclo de retroalimentación* (que consiste en observación, predicción y reacción) usado en la afinación de desempeño automática (seleccionar automáticamente el diseño físico de la base de datos para optimizar el desempeño de un sistema de base de datos relacional basado en una carga de trabajo de consultas y actualizaciones SQL). Dicha propuesta consiste en observar el cambio de la carga de trabajo para detectar un tiempo de carga de trabajo relativamente bajo, y entonces predice la carga de trabajo próxima con base en las características de la carga de trabajo actual e implementa los nuevos fragmentos verticales.

Agrawal et. al [65] integran fragmentación vertical y horizontal en una herramienta de diseño físico automático. La principal desventaja de este trabajo es que sólo recomiendan qué fragmentos deben crearse, pero el DBA es quien debe crear los fragmentos.

Autopart [67] es una herramienta automática que divide las tablas de una base de datos original de acuerdo a una carga de trabajo representativa. Autopart recibe como entrada una carga de trabajo y diseña un nuevo esquema de la base de datos usando fragmentación vertical, la desventaja principal de esta herramienta es que el DBA tiene que dar la carga de trabajo a Autopart. Sería mejor que recolectara las sentencias SQL en el momento en que se ejecuten.

A la fragmentación vertical también se le llama agrupamiento de atributos. Guinepain

y Gruenwald [101] presentan una técnica eficiente, que de manera automática y dinámica, agrupa atributos con base en los conjuntos de elementos extraídos de los conjuntos de atributos encontrados en las consultas que se ejecutan en la base de datos.

Autoclust [102] es un algoritmo de agrupamiento automático de atributos para computadoras individuales o grupos de computadoras. Este se basa en conjuntos de elementos similares encontrados en las consultas y sus atributos usando minería de reglas de asociación.

3.4. Fragmentación dinámica en bases de datos multimedia

Hasta donde sabemos actualmente no existen métodos de fragmentación dinámica para bases de datos multimedia. Las propuestas relacionadas con fragmentación dinámica en la literatura no consideran datos multimedia [14, 47, 65, 67, 100, 102]. Las propuestas que toman en cuenta datos multimedia son estáticas [16, 18, 19]. Sin embargo, una técnica de fragmentación dinámica para MMDBs permitiría mejorar el desempeño de la base de datos considerablemente, ya que sería capaz de detectar oportunamente el momento en el que el rendimiento de las consultas deje de ser adecuado. Una vez realizada esta detección se dispararía una nueva fragmentación. Esto evitaría la degradación del esquema de fragmentación y, por lo tanto, el incremento en el tiempo de respuesta de las consultas. En un ambiente multimedia distribuido las consultas cambian rápidamente con el tiempo, por tal motivo una estategia de fragmentación dinámica es más adecuada que una estática.

Capítulo 4

MAVP: Fragmentación Vertical Multimedia Adaptable

Los DBMS multimedia son capaces de almacenar y recuperar objetos compuestos por texto, imágenes, sonidos, animaciones, voz, video, etc. En los DBMS tradicionales (tanto relacionales como orientados a objetos), la principal preocupación de desempeño es la eficiencia (cuánto tiempo toma responder una consulta) [101], [88], [103]. En los DBMS multimedia, la eficiencia es todavía más importante debido al gran tamaño de los objetos multimedia [9]. Las bases de datos multimedia (MMDBs) son usualmente accedidas de manera remota a través de una red. Los objetos multimedia identificados como relevantes para la consulta deben recuperarse del servidor y transmitirse al cliente para su presentación [22].

Las técnicas de fragmentación vertical pueden aplicarse a MMDBs para proveer un acceso rápido a los objetos relevantes. Esto se debe a que en estas bases de datos los atributos tienden a ser objetos de gran tamaño y si no hay una técnica particular de almacenamiento, entonces las consultas se procesan por medio de una búsqueda secuencial de la base de datos completa, incluso aunque las consultas no requieran todos los atributos de la base de datos. Los atributos almacenados en la base de datos que son irrelevantes (esto es, no requeridos por una consulta) incrementan considerablemente el costo de procesamiento y recuperación

de las consultas, esto provoca muchos accesos a disco. Usando técnicas de fragmentación vertical podemos reducir los atributos irrelevantes accedidos por las consultas con el fin de proveer una recuperación eficiente de los objetos multimedia [15].

En MMDBs distribuidas, cuando los atributos relevantes (esto es, requeridos por las consultas) están en diferentes fragmentos y asignados a distintos sitios, hay un costo adicional debido al acceso remoto a los datos. Por lo tanto, una de las características deseables de estas bases de datos que necesita lograrse por medio de la fragmentación vertical es maximizar la accesibilidad local, esto es, cada sitio debe ser capaz de procesar las consultas localmente con acceso mínimo a los datos localizados en sitios remotos. Idealmente nos gustaría que todas las consultas accedieran a los atributos de un solo fragmento sin acceder a atributos irrelevantes en ese fragmento. Pero esto es imposible de lograrse en general porque las consultas acceden subconjuntos de atributos diferentes y superpuestos y los atributos relevantes pueden residir en sitios diferentes. Así que sólo tenemos que buscar un esquema de fragmentación vertical (EFV) que minimice la cantidad de atributos irrelevantes y atributos remotos accedidos por las consultas [73].

En este capítulo presentamos un algoritmo de fragmentación vertical para MMDBs distribuidas llamado MAVP (Multimedia Adaptable Vertical Partitioning), que toma en cuenta el tamaño de los atributos en el proceso de fragmentación vertical. La función objetivo de MAVP puede incrementar eficientemente la accesibilidad local y reducir la cantidad de accesos a atributos multimedia irrelevantes, por lo tanto, puede obtener un EFV óptimo.

4.1. Escenario

Para ilustrar y motivar nuestra propuesta, consideramos el siguiente escenario de una MMDB simple utilizada para gestionar equipos en una compañía de venta de maquinaria. La base de datos consiste en la tabla *EQUIPO (id, nombre, imagen, gráfico, audio, video)*, donde cada tupla describe información acerca de un equipo específico, incluyendo su imagen, gráfico, audio y video. Consideramos también las siguientes consultas:

q_1:Encuentra todas las imágenes y gráficos de podadoras

q_2:Encuentra nombre, audio y video con id "MB01"

q_3:Encuentra todos los gráficos, audio y video

q_4:Encuentra todas las imágenes de motobombas

4.2. Motivación

Las propuestas de fragmentación vertical sólo se basan en los atributos utilizados por las consultas y sus respectivas frecuencias para obtener un EFV. Las bases de datos tradicionales sólo contienen atributos alfanuméricos, el tamaño de diferentes atributos de este tipo es muy similar. Por ejemplo, tomamos la AUM de [36] (ver Tabla 4.1) para ilustrar que la diferencia en tamaño entre los atributos alfanuméricos no es relevante. Las entradas 0/1 en la AUM muestran si las consultas (Q) utilizan los atributos (A) o no. La columna frecuencia (F) muestra la frecuencia con la que cada consulta accede a los atributos por unidad de tiempo (por ejemplo, un día). La fila de tamaño de los atributos (S) da el número de bytes por cada atributo. Como podemos ver en la Tabla 4.1, el atributo más pequeño es a_7 con 3 bytes y el más grande es a_5 con 15 bytes, así que la diferencia entre ellos es de 12 bytes.

Sin embargo, en MMDBs el tamaño de los atributos es muy importante porque los atributos multimedia normalmente requieren una mayor cantidad de almacenamiento en disco y memoria (por ejemplo, es común que un videoclip ocupe más de 100 MB de almacenamiento en disco) que los atributos alfanuméricos. Debido a que en las MMDBs tenemos tanto atributos alfanuméricos como atributos multimedia, el tamaño de diferentes atributos es muy variado. Por ejemplo, en la AUM de la tabla EQUIPO (Tabla 4.2) podemos ver que el atributo más pequeño es *id* con 8 bytes y el más grande es *video* con 39518 bytes, así que la diferencia entre ellos es de 39510 bytes.

Para evitar la degradación del desempeño de la MMDB no es deseable almacenar objetos multimedia muy grandes, por tal motivo el administrador de la base de datos (DBA) se encarga de especificar el tamaño máximo de un atributo multimedia. Por ejemplo, en la

Tabla 4.1: Matriz de uso de atributos de una base de datos tradicional.

Q/A	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	F
q_1	1	0	0	0	1	0	1	0	0	0	25
q_2	0	1	1	0	0	0	0	1	1	0	20
q_3	0	0	0	1	0	1	0	0	0	1	25
q_4	0	1	0	0	0	0	1	1	0	0	35
q_5	1	1	1	0	1	0	1	1	1	0	25
q_6	1	0	0	0	1	0	0	0	0	0	25
q_7	0	0	1	0	0	0	0	0	1	0	25
q_8	0	0	1	1	0	1	0	0	1	1	15
S	10	8	4	6	15	14	3	5	9	12	

Tabla 4.2: Matriz de uso de atributos de una MMDB (relación EQUIPO).

Q/A	id	nombre	imagen	gráfico	audio	video	F
q_1	0	1	1	1	0	0	15
q_2	1	1	0	0	1	1	10
q_3	0	0	0	1	1	1	25
q_4	0	1	1	0	0	0	20
S	8	20	900	500	4100	39518	

Tabla 4.2 el tamaño del atributo *video* es 39518 bytes, esto significa que todos los videos de la base de datos pueden tener un tamaño máximo de 39518 bytes.

Con las propuestas de fragmentación actuales todos los atributos se consideran iguales porque dichas propuestas no toman en cuenta el tamaño de los atributos. Sin embargo, en MMDBs distribuidas no es lo mismo acceder a un atributo irrelevante *id* que a un atributo irrelevante *video* y también es diferente acceder remotamente a un atributo *id* que a un atributo *video*. Necesitamos desarrollar un EFV que tome en cuenta el tamaño de los atributos para evitar el acceso a objetos multimedia irrelevantes y la transferencia de objetos multimedia con el objetivo de obtener un mejor EFV.

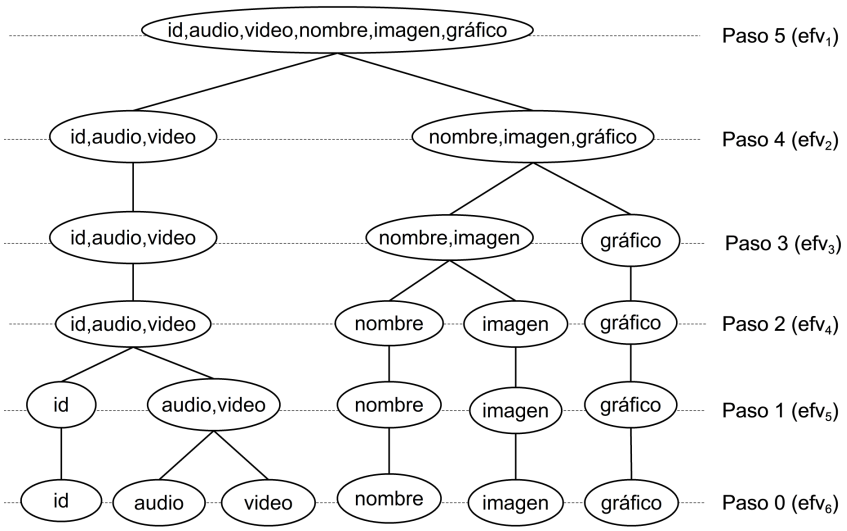


Figura 4.1: Árbol de partición de la relación EQUIPO obtenida por MAVP.

4.3. Árbol de partición

MAVP requiere como entrada la AUM y el tamaño de los atributos. Es una extensión del algoritmo AVP presentado en [74], por lo tanto, soporta fragmentación vertical n-aria, así como de mejor ajuste (óptima) de MMDBs. La primera consiste en generar un número específico de fragmentos requeridos por el usuario, la segunda es generar un EFV óptimo que minimice el costo de procesamiento de las consultas sin restricción en el número de fragmentos generados.

MAVP se basa en una propuesta ascendente. Primero, inicia con fragmentos de un solo atributo y, posteriormente, forma un nuevo fragmento seleccionando y fusionando dos fragmentos. Este proceso se repite hasta que se obtiene un fragmento compuesto por todos los atributos. Este tipo de propuesta genera un árbol binario, llamado árbol de partición (PT, en inglés: Partition Tree). La Figura 4.1 muestra el PT de la relación EQUIPO obtenido por MAVP. El PT se construye de los nodos hoja al nodo raíz. Cada nodo en un PT corresponde a un fragmento, especialmente, los nodos hoja corresponden a fragmentos de un sólo atributo y un nodo raíz corresponde a un fragmento compuesto por todos los

atributos.

Cuando se fusionan dos fragmentos se reduce la cantidad de atributos relevantes remotos accedidos mientras se incrementa la cantidad de atributos irrelevantes accedidos por las consultas. Por ejemplo, en la Tabla 4.2 tenemos que las consultas q_2 y q_3 acceden a los atributos multimedia *audio* y *video*, al inicio del proceso ellos están en diferentes fragmentos, pero si se fusionan en un fragmento, q_2 y q_3 sólo tienen que acceder a dicho fragmento, resultando en la reducción del acceso a atributos remotos.

La fusión de los atributos también causa que se incremente la cantidad de atributos irrelevantes accedidos por las consultas. Por ejemplo, si los atributos *nombre* e *imagen* se fusionan en un fragmento, la consulta q_2 sólo utiliza el atributo alfanumérico *nombre*, pero también tiene que acceder al atributo multimedia *imagen*. Por lo tanto, el fragmento fusionado incrementará la cantidad de accesos a atributos irrelevantes.

En AVP, durante cada paso en la construcción de un PT, se seleccionan dos nodos (fragmentos) que al unirse y formar un nuevo nodo maximizan el beneficio de fusión (Merging Profit) definido a continuación.

Definición 4.1

$$MergingProfit(AVP) = c \cdot d\_DF - n\_IA$$

donde

d_DF *: #DF reducido (esto es, la frecuencia total de acceder a diferentes fragmentos)*

n_IA*: #IA creado (esto es, la frecuencia total de accesos interferidos entre consultas)*

c*: una constante proporcional entre #DF y #IA*

4.4. Beneficio de fusión de MAVP

Una desventaja del beneficio de fusión de AVP es que usa una constante c, pero no se da una pauta para establecer el valor de este parámetro, ya que usando diferentes valores obtenemos diferentes EFVs, es necesario encontrar el valor adecuado de c con el fin de obtener el EFV óptimo, así que esto complica el uso del beneficio de fusión.

En nuestra función de beneficio de fusión eliminamos dicha complejidad debido a que no necesitamos ninguna constante; todos los valores se obtienen fácilmente con la AUM. Tomando en cuenta el tamaño de los atributos definimos la función de beneficio de fusión como:

Definición 4.2

$$MergingProfit(MAVP) = \#Q1 \cdot DRA - \#Q2 \cdot IIA$$

donde

#Q1: número de consultas que reducen el acceso a atributos remotos

DRA: cantidad reducida de atributos remotos accedidos

#Q2: número de consultas que incrementan el acceso a atributos irrelevantes

IIA: cantidad incrementada de atributos irrelevantes accedidos

En cada paso durante la construcción de un PT, MAVP produce un EFV fusionando dos fragmentos que maximizan la función de beneficio de fusión de la Definición 4.2, por lo tanto, cuando se termina la construcción del PT tenemos un conjunto de EFVs $EFV=\{efv_1, efv_2, \ldots, efv_n\}$, cada efv_i tiene un conjunto de fragmentos $FR=\{fr_1, fr_2, \ldots, fr_p\}$.

Para seleccionar dos fragmentos de p fragmentos que puedan maximizar el beneficio de fusión deben examinarse $\binom{p}{2} = (\frac{p(p-1)}{2})$ pares. Por ejemplo, en el Paso 0 (efv_6) p=n (donde n es el número de atributos) porque cada atributo está situado en un fragmento diferente, así que hay seis fragmentos en el Paso 0 de la Figura 4.1 y es necesario examinar el beneficio de fusión de $(\frac{6(6-1)}{2}) = 15$ pares y fusionar el par con el máximo beneficio de fusión, esto genera el efv_5 del Paso 1 en la Figura 4.1.

La tabla 4.3 muestra la matriz de beneficios de fusión (MPM, en inglés: Merging Profit Matrix) de MAVP de la tabla EQUIPO en el Paso 0. En el Algoritmo 4.1 se muestra el proceso para obtener la MPM.

MAVP se presenta en el Algoritmo 4.2, este usa como entrada la AUM de la relación con la fila de tamaño de los atributos y genera el EFV óptimo.

Tabla 4.3: Beneficios de fusion de la relación EQUIPO en el paso 0.

	id	nombre	imagen	gráfico	audio	video
id		-280	-27840	-15960	40880	395060
nombre			55400	-38700	-390800	-3755510
imagen				-44000	-700000	-5658520
gráfico					-18000	-195090
audio						3053260
video						

Algoritmo 4.1 getMPM

Entrada: AUM de la relación (un conjunto de atributos alfanuméricos y multimedia A=$\{a_1,a_2,...,a_n\}$, un conjunto de consultas Q=$\{q_1,q_2,...,q_m\}$, la frecuencia f_k de cada consulta q_k F=$\{f_1,f_2,...,f_m\}$, el tamaño s_i de cada atributo a_i S=$\{s_1,s_2,...,s_n\}$).
Salida: MPM Matriz de beneficio de fusión
begin
for each atributo $a_i \in$ A, $1 \leq i \leq n\text{-}1$ do
  for each atributo $a_j \in$ A, $i+1 \leq j \leq n$ do
   for each consulta $q_k \in$ Q, $1 \leq k \leq m$ do
     if AUM(q_k,a_i) = 1 & AUM(q_k,a_j) = 1 **then**
       #Q1 =#Q1+1
       DRA=DRA+$f_k \cdot (s_i+s_j)$
      else
       if AUM(q_k,a_i) = 1 & AUM(q_k,a_j) = 0 **then**
         #Q2=#Q2+1
         IIA=IIA+$f_k \cdot s_j$
       else
         if AUM(q_k,a_i) = 0 & AUM(q_k,a_j) = 1 **then**
           #Q2=#Q2+1
           IIA=IIA+$f_k \cdot s_i$
         end_if
       end_if
      end_if
     end_for
     merging_profit=#Q1·DRA-#Q2·IIA
     MPM(i,j)=*merging_profit*
  end_for
end_for
end. {getMPM}

Algoritmo 4.2 MAVP
Entrada: AUM
Salida: Esquema de fragmentación vertical óptimo (EFV)
begin
for each paso en PT **do**
  getMPM(AUM, MPM)
  seleccionar dos nodos con el máximo *merging_profit*
  fusionar los nodos
end_for
calcular el costo de cada paso
EFV óptimo= paso con el menor costo
end. {MAVP}

La tabla 4.4 muestra los EFVs de la tabla EQUIPO obtenidas usando AVP y MAVP.

Tabla 4.4: EFVs de la relación EQUIPO.

EFV	Algoritmos	
	AVP	MAVP
efv_1	$fr_1(a_1,a_2,a_3,a_4,a_5,a_6)$	$fr_1(a_1,a_2,a_3,a_4,a_5,a_6)$
efv_2	$fr_1(a_1,a_2,a_3)$ $fr_2(a_4,a_5,a_6)$	$fr_1(a_1,a_5,a_6)$ $fr_2(a_2,a_3,a_4)$
efv_3	$fr_1(a_1)$ $fr_2(a_2,a_3)$ $fr_3(a_4,a_5,a_6)$	$fr_1(a_1,a_5,a_6)$ $fr_2(a_2,a_3)$ $fr_3(a_4)$
efv_4	$fr_1(a_1)$ $fr_2(a_2,a_3)$ $fr_3(a_4)$ $fr_4(a_5,a_6)$	$fr_1(a_1,a_5,a_6)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$
efv_5	$fr_1(a_1)$ $fr_2(a_2,a_3)$ $fr_3(a_4)$ $fr_4(a_5)$ $fr_5(a_6)$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5,a_6)$
efv_6	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$
$a_1 = id$, $a_2 = nombre$, $a_3 = imagen$, $a_4 = gráfico$, $a_5 = audio$, $a_6 = video$		

4.5. Modelo de costo

MAVP calcula el costo de cada paso para obtener el EFV óptimo. Fue necesario modificar el modelo de costo de AVP para adaptarlo a MMDBs distribuidas, ya que dicho modelo no toma en cuenta el tamaño de los atributos. MAVP seleccionará el EFV que minimice el acceso a atributos irrelevantes y maximice la accesibilidad local (minimice el costo de trasporte) de las consultas.

MAVP produce un EFV en cada paso en la construcción del PT, así que cuando se finaliza el PT tenemos un conjunto de EFVs $EFV=\{efv_1, efv_2, \ldots, efv_n\}$, cada efv_i tiene

un conjunto de fragmentos efv_i=*{*fr_1, fr_2, ... , fr_p*}*, y cada fragmento fr_k tiene n_k atributos, suponiendo que la red tiene los nodos N_1, ... , N_p, la asignación de los fragmentos a los nodos ocasiona un mapeo λ: *{1,...,p}* $\rightarrow$ *{1,...,p}* llamado asignación de ubicación [13]. Para determinar el efv_i óptimo, necesitamos evaluar como afecta la fragmentación vertical a los costos totales de las consultas, para esto utilizamos la AUM. El costo de efv_i se compone de dos partes: el costo de acceso a atributos irrelevantes y el costo de transporte.

$$cost(efv_i) = IAAC(efv_i) + TC(efv_i) \tag{4.1}$$

El costo de acceso a atributos irrelevantes mide la cantidad de atributos irrelevantes accedidos durante una consulta. El costo de transporte provee una medida de transporte entre los nodos de la red.

El costo de acceso a atributos irrelevantes esta dado por

$$IAAC(efv_i) = \sum_{k=1}^{p} IAAC(fr_k) \tag{4.2}$$

La AUM de una relación tiene un conjunto de atributos alfanuméricos y multimedia A=*{*a_1,a_2,...,a_n*}*, el tamaño s_i de cada atributo a_i S=*{*s_1,s_2, ...,s_n*}*, un conjunto de consultas Q=*{*q_1,q_2, ...,q_m*}*, la frecuencia f_j de cada consulta q_j F=*{*f_1,f_2, ...,f_m*}*, un conjunto de elementos u_{ji}, donde u_{ji}=*1* si la consulta q_j usa el atributo a_i, o si no u_{ji}=*0*. El costo de acceso a atributos irrelevantes de cada fragmento fr_k esta dado por

$$IAAC(fr_k) = \begin{cases} \sum\limits_{q_j \in P_k} f_j \sum\limits_{i|a_i \in fr_k \wedge u_{ji}=0} s_i & \text{si } n_k > 1 \\ 0 & \text{de otra manera} \end{cases} \tag{4.3}$$

Donde P_k es un conjunto de consultas que usan al menos un atributo y acceden al menos un atributo irrelevante del fragmento fr_k. Esto es

$$P_k = \{q_j | u_{ji} = 0 \wedge u_{jl} = 1, 1 \leq i, l \leq n_k\} \tag{4.4}$$

Por ejemplo, IAAC(efv_3) de AVP en la Tabla 4.4 se calcula como sigue.

IAAC(efv$_3$) = IAAC(fr$_1$) + IAAC(fr$_2$) + IAAC(fr$_3$). IAAC(fr$_1$)=0 porque n_1=1, P_1=$\{\varnothing\}$ debido a que sólo hay un atributo en el fragmento fr_1. P_2=$\{q_2\}$ porque q_2 sólo usa el atributo alfanumérico *nombre* pero también tiene que acceder al atributo multimedia irrelevante *imagen* debido a que ambos están ubicados en el mismo fragmento. Por lo tanto, *IAAC(fr$_2$)=10·900=9000*, porque f_2=10 y s_{imagen}=900. P_3=$\{q_1, q_2\}$, ya que q_1 sólo utiliza el atributo multimedia *gráfico* pero también tiene que acceder a los atributos multimedia irrelevantes *audio* y *video* debido a que están ubicados en el mismo fragmento. Por otro lado, q_2 usa los atributos *audio* y *video* pero tiene que acceder al atributo multimedia irrelevante *gráfico*, así que *IAAC(fr$_3$)=15·(4100+39518)+10·(500)=659270*, porque f_1=15, f_2=10, s_{audio}=4100, s_{video}=39518 y $s_{gráfico}$=500. Por lo que, *IAAC (efv$_3$)*=0+9000+659270=668270.

Dada una asignación de ubicación podemos calcular el costo de trasporte de un efv_i. El costo de transporte de efv_i es la suma de los costos de cada consulta multiplicada por su frecuencia al cuadrado, esto es.

$$TC(efv_i) = \sum_{j=1}^{m} TC(q_j).f_j^2 \tag{4.5}$$

El costo de transporte de la consulta q_j depende del tamaño de los atributos remotos relevantes y de las ubicaciones asignadas que deciden el costo de transporte entre cada par de sitios. Esto puede expresarse por

$$TC(q_j) = \sum_{h} \sum_{h'} c_{\lambda(h)\lambda(h')}.s(h') \tag{4.6}$$

Donde h varía entre los nodos de la red, *s(h')* son los tamaños de los atributos remotos relevantes, *λ(h)* indica el nodo en la red en el que se almacena la consulta, y c_{ij} es un factor de costo de transporte del nodo N_i al nodo N_j *(i, j ∈ {1, . . . , p})* [13].

Por ejemplo, TC(efv_3) de AVP en la Tabla 4.4 se calcula como sigue. Hay tres fragmentos, así que suponemos que hay tres nodos N_1, N_2, N_3 y cada fragmento fr_i se localiza

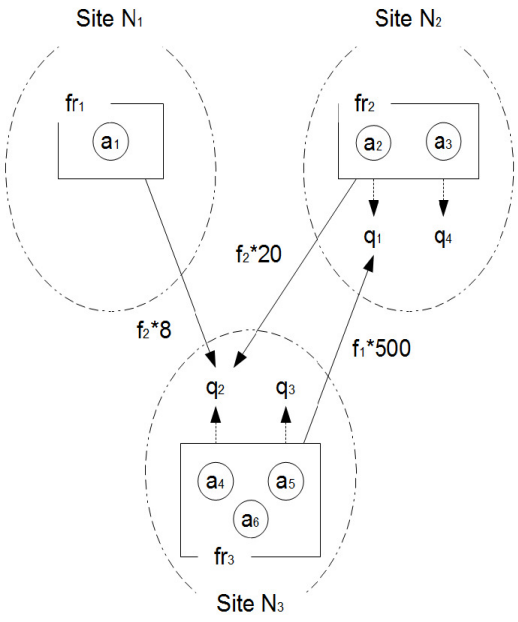


Figura 4.2: Asignación de ubicación de efv$_3$.

en cada nodo N_i, también asumimos que cada consulta se localiza en el nodo donde están ubicados los atributos de mayor tamaño que utiliza y que c_{ij} =1. En la Figura 4.2 se presenta la asignación de ubicación del efv_3.

TC(efv_3)=($s_{gráfico}$*15^2)+((s_{id}+s_{nombre})*10^2)=115300 porque q_1 accede el atributo multimedia remoto *gráfico* y q_2 accede los atributos alfanuméricos remotos *id* y *nombre*. Por lo tanto, $cost(efv_3)$=668270+115300=783570.

4.6. Experimentos y evaluación

En esta sección, se presentan algunos resultados experimentales para evaluar la eficiencia del algoritmo MAVP. En estos experimentos se utiliza como base el modelo de costo propuesto para comparar nuestro algoritmo MAVP con AVP de Son y Kim [74]. Para el primer experimento, considerando la información de la AUM de la relación EQUIPO presentada en la Tabla 4.2, se calculó el costo de los EFVs obtenidos por AVP y MAVP de la Tabla 4.4 y los resultados se muestran en la Tabla 4.5.

Tabla 4.5: Comparación de costo del experimento I.

EFV	AVP			MAVP		
	IAAC	TC	Costo	IAAC	TC	Costo
efv_1	1574110	0	1574110	1574110	0	1574110
efv_2	668550	115300	783850	47200	314500	361700
efv_3	668270	115300	783570	9200	427000	436200
efv_4	9000	427800	436800	200	439500	439700
efv_5	9000	3400300	3409300	0	440300	440300
efv_6	0	3412800	3412800	0	3412800	3412800

Como podemos ver en la Tabla 4.5, los EFVs generados con MAVP tienen un costo más bajo que los obtenidos con AVP en la mayoría de los casos, esto se debe a que AVP sólo considera información acerca de las consultas con respecto a sus atributos utilizados y su frecuencia en el proceso de fragmentación vertical, mientras que MAVP también toma en cuenta el tamaño de los atributos, usando esta información MAVP puede evitar acceder a atributos multimedia irrelevantes de gran tamaño, reduciendo el costo de ejecución de las consultas considerablemente.

Los esquemas primeros y últimos (efv_1 *y* efv_6) son iguales para ambos algoritmos. En efv_1 (tabla no fragmentada) el costo de transporte es cero y el costo de acceso a atributos irrelevantes es máximo, la razón de esto es que todos los atributos están ubicados en el mismo fragmento y las consultas no acceden a ningún atributo remoto, pero ellas recorren toda la tabla con el fin de obtener los atributos relevantes y esto provoca el acceso a muchos atributos irrelevantes. Por otro lado, cuando el número de fragmentos es igual al número de atributos de la relación (efv_6), el costo de transporte es máximo y el costo de acceso a atributos irrelevantes es cero, porque cada atributo se ubica en un fragmento diferente, como resultado, una consulta accede a muchos atributos remotos pero a ningún atributo irrelevante.

El costo de atributos irrelevantes accedidos por las consultas (IAAC) se reduce considerablemente en los otros esquemas obtenidos por MAVP (efv_2 a efv_5), ya que AVP se enfoca en la minimización del acceso a diferentes fragmentos y este factor sólo implica un costo de transporte más bajo, MAVP también considera la minimización de atributos

irrelevantes accedidos por las consultas, este factor es muy importante en MMDBs porque los atributos tienden a ser objetos de gran tamaño y usualmente las consultas acceden sólo subconjuntos de los atributos en una relación, por lo tanto, si se reducen los atributos multimedia irrelevantes, se obtiene una reducción importante en el costo de ejecución de las consultas.

AVP encuentra el EFV óptimo cuando el número de fragmentos es igual a tres, esto es, efv_3=*{*fr_1=*(id), *fr_2=*(nombre, imagen), *fr_3=*(gráfico, audio, video)}*, pero con el modelo de costo que sólo toma en cuenta el costo de transporte. Sin embargo, con nuestro modelo de costo se determinó que la solución óptima de los esquemas de AVP es efv_4=*{*fr_1 =(*id*), fr_2 =(*nombre, imagen*), fr_3 =(*gráfico*), fr_4 =(*audio, video*)*}* con un costo de 436800. MAVP obtuvo una mejor solución cuando el número de fragmentos es igual a dos efv_2=*{*fr_1=*(id, audio, video), *fr_2=*(nombre, imagen, gráfico)}*, con un costo de 361700, este EFV tiene el costo más bajo.

Tabla 4.6: Comparación de costo del experimento II.

EFV	AVP			MAVP		
	IAAC	TC	Costo	IAAC	TC	Costo
efv_1	13065	0	13065	13065	0	13065
efv_2	5745	2925	8670	5745	2925	8670
efv_3	1925	6600	8525	1925	6600	8525
efv_4	975	25975	26950	975	25975	26950
efv_5	0	59100	59100	0	59100	59100
efv_6	0	69300	69300	0	64200	64200
efv_7	0	74400	74400	0	74400	74400
efv_8	0	93150	93150	0	80525	80525
efv_9	0	99275	99275	0	99275	99275
efv_{10}	0	111775	111775	0	111775	111775

En el segundo experimento se usó la AUM de una base de datos tradicional de la Tabla 4.1, que se utilizó en Navathe et al. [36]. La comparación de costo de los esquemas encontrados con AVP y MAVP se presenta en la Tabla 4.6. En este caso la mayoría de los EFVs generados por MAVP son iguales que los de AVP y ambos algoritmos obtuvieron la misma solución óptima (esto es, efv_3=*{*fr_1=*(*a_1, a_5, a_7*), *fr_2=*(*a_2, a_3, a_8, a_9*), *fr_3=*(*a_4,

a_6, a_{10}*)})* porque el tamaño de los atributos es muy similar. Sin embargo, efv_6 y efv_8 son diferentes para ambos algoritmos y MAVP encontró un esquema con menor costo en ambos casos. Se modificó el tamaño de los atributos y MAVP generó los EFVs de la Tabla 4.7.

Tabla 4.7: Fragmentos resultantes del experimento II con diferente tamaño.

EFV	Algoritmos	
	AVP	MAVP
efv_1	$fr_1(a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9,a_{10})$	$fr_1(a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9,a_{10})$
efv_2	$fr_1(a_1,a_2,a_3,a_5,a_7,a_8,a_9)$ $fr_2(a_4,a_6,a_{10})$	$fr_1(a_1,a_2,a_3,a_5,a_7,a_8,a_9)$ $fr_2(a_4,a_6,a_{10})$
efv_3	$fr_1(a_1,a_5,a_7)$ $fr_2(a_2,a_3,a_8,a_9)$ $fr_3(a_4,a_6,a_{10})$	$fr_1(a_1,a_5,a_7)$ $fr_2(a_2,a_3,a_8,a_9)$ $fr_3(a_4,a_6,a_{10})$
efv_4	$fr_1(a_1,a_5)$ $fr_2(a_2,a_3,a_8,a_9)$ $fr_3(a_4,a_6,a_{10})$ $fr_4(a_7)$	$fr_1(a_1)$ $fr_2(a_2,a_3,a_8,a_9)$ $fr_3(a_4,a_6,a_{10})$ $fr_4(a_5,a_7)$
efv_5	$fr_1(a_1,a_5)$ $fr_2(a_2,a_8)$ $fr_3(a_3,a_9)$ $fr_4(a_4,a_6,a_{10})$ $fr_5(a_7)$	$fr_1(a_1)$ $fr_2(a_2,a_3,a_8,a_9)$ $fr_3(a_4,a_6,a_{10})$ $fr_4(a_5)$ $fr_5(a_7)$
efv_6	$fr_1(a_1,a_5)$ $fr_2(a_2,a_8)$ $fr_3(a_3,a_9)$ $fr_4(a_4,a_6)$ $fr_5(a_7)$ $fr_6(a_{10})$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3,a_9,a_8)$ $fr_4(a_4,a_6,a_{10})$ $fr_5(a_5)$ $fr_6(a_7)$
efv_7	$fr_1(a_1,a_5)$ $fr_2(a_2,a_8)$ $fr_3(a_3,a_9)$ $fr_4(a_4)$ $fr_5(a_6)$ $fr_6(a_7)$ $fr_7(a_{10})$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3,a_9)$ $fr_4(a_4,a_6,a_{10})$ $fr_5(a_5)$ $fr_6(a_7)$ $fr_7(a_8)$
efv_8	$fr_1(a_1)$ $fr_2(a_2,a_8)$ $fr_3(a_3,a_9)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$ $fr_7(a_7)$ $fr_8(a_{10})$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4,a_6,a_{10})$ $fr_5(a_5)$ $fr_6(a_7)$ $fr_8(a_8)$ $fr_8(a_9)$
efv_9	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3,a_9)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$ $fr_7(a_7)$ $fr_8(a_8)$ $fr_9(a_{10})$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6,a_{10})$ $fr_7(a_7)$ $fr_8(a_8)$ $fr_9(a_9)$
efv_{10}	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$ $fr_7(a_7)$ $fr_8(a_8)$ $fr_9(a_9)$ $fr_{10}(a_{10})$	$fr_1(a_1)$ $fr_2(a_2)$ $fr_3(a_3)$ $fr_4(a_4)$ $fr_5(a_5)$ $fr_6(a_6)$ $fr_7(a_7)$ $fr_8(a_8)$ $fr_9(a_9)$ $fr_{10}(a_{10})$
$s_1 = 8$, $s_2 = 20$, $s_3 = 20$, $s_4 = 15$, $s_5 = 10$, $s_6 = 250$ $s_7 = 900$, $s_8 = 500$, $s_9 = 4100$, $s_{10} = 39518$		

La comparación de costo de los esquemas obtenidos con AVP y MAVP se presenta en la Tabla 4.8. Los esquemas de AVP permanecen estáticos cuando cambia el tamaño de los atributos. Sin embargo, MAVP adapta los esquemas de acuerdo a los cambios en el tamaño de los atributos. Por lo tanto, los esquemas y la solución de mejor ajuste (óptima) obtenida por AVP son las mismas incluso cuando se cambió el tamaño de los atributos. MAVP encontró diferentes esquemas, en este caso sólo los primeros tres esquemas son iguales para ambos algoritmos. Usando el modelo de costo de AVP la mejor solución es todavía efv_3, pero con nuestro modelo de costo se obtuvo como solución óptima efv_2=*{*fr_1=*(*a_1,a_2,a_3,a_5,a_7,a_8,a_9*),* fr_2=*(*a_4,a_6,a_{10}*)}*. En este caso MAVP encontró la misma solución óptima, pero de nuevo

Tabla 4.8: Comparación de costo del experimento II con diferente tamaño.

EFV	AVP			MAVP		
	IAAC	TC	Cost	IAAC	TC	Cost
efv_1	8001555	0	8001555	8001555	0	8001555
efv_2	502750	927000	1429750	502750	927000	1429750
efv_3	188130	2137750	2325880	188130	2137750	2325880
efv_4	165000	2149000	2314000	187850	2147750	2335600
efv_5	0	3774000	3774000	165000	2154000	2319000
efv_6	0	3999250	3999250	164200	2216500	2380700
efv_7	0	3999250	3999250	0	3779000	3779000
efv_8	0	4004250	4004250	0	3854000	3854000
efv_9	0	4004250	4004250	0	3866750	3866750
efv_{10}	0	4079250	4079250	0	4079250	4079250

MAVP generó EFVs con menor costo en más casos (efv_5 a efv_9).

4.7. Discusión

Se comparó MAVP contra AVP y se obtuvieron EFVs con menor costo en más casos usando MAVP. Por lo tanto, el tamaño de los atributos es un factor muy importante a considerar en el proceso de fragmentación vertical con el fin de encontrar una solución óptima en MMDBs distribuidas. Las principales ventajas de MAVP sobre otras propuestas son las siguientes:

1. La mayoría de los algoritmos de fragmentación vertical [88], [15], [73], [68], [37], [104], [71], [74] no consideran el tamaño de los atributos como entrada al proceso de fragmentación vertical. Por lo tanto, dichos algoritmos no son adecuados para MMDBs donde el tamaño de los atributos es muy variado. MAVP toma en cuenta el tamaño de los atributos para generar esquemas de fragmentación vertical que reducen considerablemente el costo de procesamiento de consultas.

2. MAVP encuentra la solución óptima usando un modelo de costo basado en la minimización del acceso a atributos irrelevantes (reduciendo accesos a disco) y del acceso a atributos remotos (reduciendo el costo de transporte), mientras que el modelo de

costo de otras propuestas de fragmentación vertical [15], [21] sólo considera la reducción del número de accesos a disco, ignorando el costo de transporte que es el costo más importante en las MMDBs distribuidas.

4.8. Conclusión

La fragmentación vertical puede proveer recuperación eficiente de objetos multimedia en MMDBs distribuidas. En este Capítulo se describió MAVP, que es un algoritmo de fragmentación vertical para MMDBs distribuidas que toma en cuenta el tamaño de los atributos para generar un EFV óptimo. Además, también se propuso un modelo de costo para MMDBs distribuidas, el cual considera que el costo de procesamiento de las consultas en un ambiente multimedia distribuido se compone de costo de acceso a atributos irrelevantes y costo de transporte. Una evaluación experimental muestra que nuestro algoritmo supera a AVP en la mayoría de los casos.

Se asume en este Capítulo que las consultas que se ejecutan en la MMDB son estáticas. Las MMDBs son usualmente accedidas por muchos usuarios simultáneamente, por lo tanto, las consultas tienden a cambiar con el tiempo y un buen EFV puede degradarse resultando en tiempo de respuesta de consultas muy lento. MAVP puede extenderse para derivar una fragmentación vertical dinámicamente en MMDBs, basada en los cambios en las consultas. Por lo tanto, el EFV de la MMDB puede adaptarse a las consultas para lograr una recuperación eficiente de objetos multimedia todo el tiempo. En los siguientes Capítulos se presentan nuestras técnicas de fragmentación vertical dinámica para bases de datos relacionales y para MMDBs.

Capítulo 5

DYVEP: Un sistema Activo de Fragmentacion Vertical Dinámica para Bases de Datos Relacionales

La fragmentación vertical se ha estudiado ampliamente en bases de datos relacionales para mejorar el tiempo de respuesta de las consultas [36], [24], [101]. En la fragmentación vertical, una tabla se divide en un conjunto de fragmentos, cada uno con un subconjunto de atributos de la tabla original y definido por un esquema de fragmentación vertical (EFV). Los fragmentos consisten de registros más pequeños, por lo tanto, menos páginas de memoria secundaria son accedidas para procesar consultas que recuperan o actualizan sólo algunos atributos de la tabla, en vez del registro completo [101].

La fragmentación vertical puede ser estática o dinámica [47]. En la fragmentación vertical estática, los atributos sólo se asignan a los fragmentos cuando se crean. Este tipo de fragmentación tiene los siguientes problemas:

1. El DBA tiene que observar el sistema por una cantidad importante de tiempo hasta que descubra las probabilidades de que las consultas accedan los atributos de la base de datos y sus frecuencias antes de que pueda realizar la operación de fragmentación,

a esta operación de observación del sistema se le conoce como etapa de análisis de requerimientos.

2. Incluso después que se completa el proceso de fragmentación, nada garantiza que se hayan descubierto las tendencias en las consultas y en los datos reales. Por lo tanto, el EFV puede no ser bueno. En este caso, los usuarios de la base de datos pueden experimentar tiempos de respuesta de consultas muy lentos.

3. Algunas aplicaciones como sistemas multimedia, de comercio electrónico, de soporte de decisiones y de información geográfica son accedidos por muchos usuarios simultáneamente. Por lo tanto, las consultas y sus respectivas frecuencias tienden a cambiar con el tiempo y un EFV se implementa para optimizar el tiempo de respuesta de un conjunto particular de consultas. Así que si las consultas o sus relativas frecuencias cambian, el EFV puede dejar de ser adecuado.

4. En los métodos de fragmentación vertical estática, realizar una nueva fragmentación es una tarea pesada y sólo puede ejecutarse manualmente cuando el sistema no se está utilizando.

En contraste, en la fragmentación vertical dinámica, los atributos se reubicarán si se determina que el EFV implementado se ha vuelto inadecuado debido a un cambio en la información de las consultas. La mayoría de los métodos de fragmentación vertical son estáticos. Es más efectivo para una base de datos verificar dinámicamente si un EFV sigue siendo adecuado para determinar en qué momento es necesaria una nueva fragmentación [105].

Los sistemas activos en bases de datos son capaces de responder automáticamente a eventos que están tomando lugar ya sea dentro o fuera del sistema. La parte central de dichos sistemas son las reglas activas que codifican el conocimiento de los expertos de dominio [106]. Las reglas activas constantemente monitorizan los sistemas y las actividades de los usuarios. Cuando pasa un evento interesante, ellas responden ejecutando ciertos procedimientos relacionados al sistema o al ambiente [107].

La mayoría de las técnicas dinámicas de agrupamientos de atributos [99], [108], [109] consisten de los siguientes módulos: un recolector de estadísticas (SC, en inglés: Statistic Collector) que acumula información acerca de las consultas ejecutadas y los datos regresados. El SC se encarga de recolectar, filtrar y analizar las estadísticas. Es responsable de disparar el Analizador de Grupos (CA, en inglés: Cluster Analyzer). El CA determina el mejor agrupamiento posible dadas las estadísticas recolectadas. Si el nuevo agrupamiento es mejor que el actual, entonces el CA dispara el Reorganizador que cambia la organización física de los datos en disco [24].

En las técnicas de fragmentación vertical dinámica es necesario monitorizar la base de datos para determinar cuándo disparar el CA y el reorganizador.

Actualmente no existen propuestas de fragmentación vertical dinámica que usen reglas activas. Sin embargo, la fragmentación vertical dinámica puede implementarse efectivamente como un sistema activo, debido a que las reglas activas son lo suficientemente expresivas para permitir la especificación de una gran clase de tareas de monitorización y no tienen impacto notable en el desempeño, particularmente cuando el sistema está bajo carga pesada. Las reglas activas son fáciles de implementar con bajos costos de memoria y CPU [25].

En este Capítulo se describe nuestra propuesta de un sistema activo de fragmentación vertical dinámica para bases de datos relacionales llamado DYVEP (DYnamic VErtical Partitioning). Usando reglas activas DYVEP puede monitorizar consultas ejecutadas en la base de datos con el fin de acumular la información necesaria para desarrollar la fragmentación vertical, además DYVEP cambia automáticamente los fragmentos de acuerdo a los cambios en los patrones de consultas y el esquema de la base de datos, logrando reducir el tiempo de respuesta de las consultas.

5.1. Reglas activas

Los sistemas activos tienen la habilidad de detectar eventos y responder a ellos automáticamente de manera oportuna [110]. Este comportamiento reactivo se especifica por

medio de reglas activas o reglas Evento - Condición- Acción (ECA). Las reglas activas determinan cuándo y cómo reaccionar a diferentes tipos de eventos. La forma general de una regla ECA es la siguiente:

`ON` *evento*

`IF` *condición*

`THEN` *acción*

Un evento es algo que ocurre en un punto de tiempo, por ejemplo, una consulta en una base de datos. La condición examina el contexto en el que toma lugar el evento. La acción describe la tarea que va a ser ejecutada por la regla si la condición se cumple una vez que toma lugar el evento. Varias aplicaciones como casas inteligentes, bases de datos activas y bases de datos de sensores integran reglas activas para el manejo de algunas de sus actividades importantes [111]. A continuación se describe cada elemento de una regla activa:

5.1.1. Evento

Un evento especifica que causa que se dispare la regla. Los eventos puede ser [112]:

- **Modificación de datos**. En un sistema de base de datos relacional, un evento de modificación de datos puede declararse como una de las tres operaciones de modificación de datos SQL - **insert**, **delete**, o **update** - en una tabla particular.

- **Recuperación de datos**. En un sistema de base de datos relacional, un evento de recuperación de datos puede determinarse como una operación **select** en una tabla particular.

- **Tiempo**. Un evento temporal puede especificar que una regla se dispare en un tiempo absoluto (por ejemplo, el 16 de septiembre de 2012 a las 12:00), en un tiempo repetido (por ejemplo, cada día a las 12:00) o en intervalos periódicos (por ejemplo, cada 10 minutos).

- **Definido por la aplicación**. Estos eventos pueden especificarse permitiendo que la aplicación declare un evento E (por ejemplo, temperatura alta, datos muy grandes, etc.) y permitiendo que las reglas especifiquen E como el evento que las dispare. Entonces, cada vez que una aplicación notifique al sistema de base de datos la ocurrencia del evento E, se dispara cualquier regla cuyo evento sea E. Usando esta propuesta, la aplicación puede ejecutar cualquier monitorización o cálculo que desee para detectar cuando debe ocurrir el evento E, esto es, detectar cuándo deben dispararse las reglas asociadas con E.

Los eventos anteriores son eventos simples o primitivos. Los eventos también pueden ser compuestos, en este caso se originan por alguna combinación de eventos primitivos o eventos compuestos. Los operadores útiles para combinar eventos son [112]:

- **Operadores lógicos**. Los eventos pueden combinarse usando operadores booleanos *and*, *or*, *not*, etc.
- **Secuencia**. Una regla podría dispararse cuando dos o más eventos ocurren en un orden particular.
- **Composición temporal**. Una regla podría dispararse por una combinación de eventos temporales y no temporales, como "*10 segundos después del evento* E_1" o "*cada día después de la primera ocurrencia del evento* E_2".

5.1.2. Condición

Cuando se detecta un evento se dispara una regla, una vez que se dispara la regla esta puede activarse o no dependiendo del resultado de evaluación de la condición. La condición de las reglas activas actúa como un filtro [113]. Las condiciones pueden ser:

- **Predicados de la base de datos**. La condición puede especificar que se cumpla un cierto predicado en la base de datos, donde el predicado se define usando un lenguaje correspondiente a las cláusulas de condición en el lenguaje de consulta de la base

de datos (por ejemplo, un sistema de base de datos relacional puede permitir como condiciones de reglas cualquier cosa correspondiente a una cláusula **where** de SQL). Por ejemplo, una condición expresada como un predicado podría declarar que el valor promedio en algún conjunto de valores este bajo un cierto umbral.

- **Predicados restringidos**. La condición puede especificar que se cumpla un cierto predicado en la base de datos, donde el predicado se define usando una porción restringida de las cláusulas de condición del lenguaje de consultas. Por ejemplo, un sistema de base de datos relacional puede restringir sus condiciones de reglas de tal forma que se permitan las operaciones de comparación pero no las operaciones agregadas.

- **Consultas de la base de datos**. La condición puede especificar una consulta usando el lenguaje de consultas del sistema de la base de datos. Por ejemplo, una condición especificada como una consulta puede recuperar todos los datos cuyos valores estén por debajo de cierto umbral y la condición es verdadera si y sólo si la consulta produce una respuesta vacía.

- **Procedimientos de aplicación**. Una condición en una regla puede especificarse como una llamada a un procedimiento escrito en un lenguaje de programación de aplicaciones, donde el procedimiento puede acceder o no acceder a la base de datos. Si el procedimiento regresa *true* entonces la condición se cumple; si el procedimiento regresa *false* entonces la condición no se cumple. También los procedimientos pueden regresar datos en lugar de un valor booleano. Esto puede significar que la condición sea verdadera si y sólo si el procedimiento regresa datos o que la condición sea verdadera si el procedimiento no regresa datos.

En las reglas activas, la condición es generalmente opcional. Si se omite la condición de una regla, el evento que dispara la regla siempre la activará.

5.1.3. Acción

La acción se ejecuta cuando se dispara la regla y su condición es verdadera. Las acciones pueden ser:

- **Operaciones de modificación de datos**. Un sistema de base de datos relacional puede permitir acciones de reglas para especificar operaciones SQL **insert**, **delete** o **update**.
- **Operaciones de recuperación de datos**. Un sistema de base de datos relacional puede permitir acciones de reglas para especificar operaciones SQL **select**.
- **Otros comandos de la base de datos**. Una acción de una regla puede permitir especificar cualquier operación de la base de datos. Además de las operaciones de modificación y recuperación, la mayoría de los sistemas de base de datos soportan operaciones para definición de datos, control de transacciones (por ejemplo, commit, rollback), operaciones para conceder y revocar privilegios, etc.
- **Procedimientos de aplicación**. Una acción de una regla puede especificarse como una llamada a un procedimiento escrito en un lenguaje de programación de aplicaciones, donde el procedimiento puede acceder o no acceder a la base de datos.

Una regla activa puede definir un conjunto de acciones, usualmente múltiples acciones se ejecutan en un orden secuencial.

5.1.4. Ejecución de reglas activas

En tiempo de ejecución una regla puede estar en una de las siguientes fases [114]:

- **Fase de señalización**: Se refiere a la ocurrencia de un evento.
- **Fase de disparo:** Toma los eventos producidos y dispara las reglas correspondientes. Si una regla se asocia con la ocurrencia de un evento, se instancia la regla.

- **Fase de evaluación:** Verifica la condición de las reglas disparadas, se forma un conjunto de reglas conflicto de todas las instancias de reglas cuyas condiciones son satisfechas.
- **Fase de calendarización:** Indica la manera en que se va a procesar el conjunto de reglas conflicto.
- **Fase de ejecución:** Lleva a cabo las acciones de las instancias de reglas elegidas. Durante la ejecución de las acciones pueden señalarse otros eventos, esto puede provocar el disparo en cascada de nuevas reglas.

5.2. Arquitectura de DYVEP

Con el fin de obtener en todo tiempo un buen desempeño de consultas se desarrolló el sistema activo para fragmentación vertical dinámica de bases de datos relacionales DYVEP. La principal ventaja de DYVEP es que permite adaptar el EFV de acuerdo a los cambios en los patrones de acceso a la base de datos, para esto utiliza una base de reglas activas, las cuales se disparan cada vez que se realizan cambios en las estadísticas de la base de datos. DYVEP recolecta estadísticas sobre los patrones de acceso y del esquema de la base de datos y obtiene un EFV de acuerdo a las estadísticas recolectadas. Usando reglas activas DYVEP puede reaccionar a los eventos generados por usuarios o procesos, evaluar condiciones y si las condiciones son verdaderas, entonces ejecutar las acciones o procedimientos definidos.

La arquitectura de DYVEP se muestra en la Figura 5.1. DYVEP se compone de 3 módulos: *Recolector de Estadísticas*, *Algoritmo de Fragmentación Vertical (VPA)* y *Generador de Fragmentos*.

5.2.1. Recolector de Estadísticas

El *Recolector de Estadísticas* acumula información importante acerca de las consultas (como identificador, descripción, atributos usados, frecuencia de acceso) y de los atributos (nombre, orden). Cuando DYVEP se ejecuta por primera vez en la base de datos,

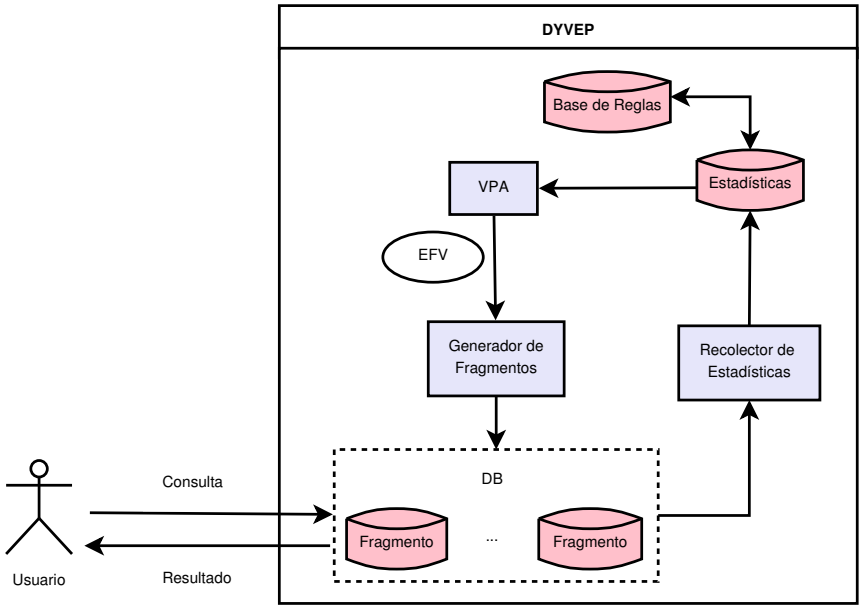


Figura 5.1: Arquitectura de DYVEP.

el *Recolector de Estadísticas* creará las tablas: **attributes (AT)**, **queries (QT)**, **attribute_usage_table (AUT)**, **attribute_affinity_table (AAT)**, **clustered_affinity_table (CAT)** y un conjunto de reglas activas en dichas tablas, las cuales se presentan a lo largo de este Capítulo.

Después de inicializar DYVEP el recolector de estadísticas obtiene las consultas ejecutadas en la base de datos y su frecuencia cada determinado tiempo (por ejemplo, cada día), para esto cuando una consulta (q_i) se ejecuta en la base de datos, el *Recolector de Estadísticas* verificará que la consulta no está almacenada en **QT**, en tal caso asignará un identificador a la consulta, almacenará su descripción y fijará su frecuencia en 1 en **QT**. Si la consulta ya se encuentra almacenada en **QT**, entonces solo se incrementará su frecuencia en 1. Esto se define en la siguiente regla activa:

insert_queries

`ON` $\mathrm{q}_i \in \mathbf{Q}$

`IF` $\mathrm{q}_i \notin \mathbf{QT}$

`THEN` insert into **QT**(*id, query, freq*) values (*id_ q_i*,*query_ q_i, 1*)

`ELSE` update **QT** set *freq*=*old.freq*+1 where *id*=*id_ q_i*

Con el fin de saber si la consulta ya se encuentra almacenada en **QT**, el *Recolector de Estadísticas* tiene que analizar las consultas. Dos consultas se consideran iguales si usan los mismos atributos, por ejemplo si tenemos las consultas:

q_1: `SELECT A, B FROM T`

q_2: `SELECT SUM(B)FROM T WHERE A=Value`

Si q_1 ya está almacenada en **QT** y q_2 se ejecuta en la base de datos, el *Recolector de Estadísticas* analizará q_2 con el fin de saber que atributos usa la consulta, y comparará q_2 con las consultas ya almacenadas en **QT**, debido a que q_1 usa los mismos atributos, entonces su frecuencia se incrementá en 1. Cuando todas las consultas ejecutadas en la base de datos se almacenan en **QT**,entonces el *Recolector de Estadísticas* dispara *VPA*.

5.2.2. Algoritmo de Fragmentación Vertical (VPA)

El *Algoritmo de Fragmentación Vertical* determina el mejor EFV de acuerdo a las estadísticas recolectadas, se presenta en el Algoritmo 5.1.

Algoritmo 5.1 VPA
Entrada:
QT: queries
Salida:
Esquema de Fragmentación Vertical óptimo (EFV)
begin
{Paso 1: Generar la **AUT**}
  getAUT(**QT**, **AUT**)
  {genera la **AUT** de la **QT**}
{Paso 2: Obtener el EFV óptimo}
  getVPS(**AUT**, EFV)
  {Obtiene el EFV óptimo usando la **AUT** generada en el Paso 1}
end. {VPA}

En el primer paso *VPA* genera la **AUT** de la tabla **queries**. En el segundo paso la **AUT** generada es utilizada por *VPA* para obtener el EFV óptimo. Finalmente, *VPA* dispara el *Generador de Fragmentos.*

5.2.3. Generador de Fragmentos

El *Generador de Fragmentos* materializa el nuevo EFV, para esto elimina los fragmentos anteriores y crea los nuevos fragmentos. Este implementa los fragmentos como índices, de esta forma el DBMS usa automáticamente los fragmentos cuando se ejecuta una consulta y no es necesario cambiar la definición de las consultas para usar los fragmentos en lugar de la tabla completa. Así que el uso de los fragmentos es transparente para los usuarios.

5.3. Implementación de DYVEP

Implementamos DYVEP usando triggers dentro del DBMS objeto-relacional de licencia libre y código abierto PostgreSQL ejecutándose en un sólo procesador Intel (R) Core(TM) i7CPU 2.67-GHz con 4 GB de memoria principal y un disco duro de 698-GB. Utilizamos el benchmark TPC-H [115].

5.3.1. Benchmark TPC-H

TPC-H es un benchmark de soporte de decisiones ampliamente utilizado en la evaluación del desempeño de los sistemas de bases de datos relacionales. Este consiste de una suite de negocios orientada a consultas y modificaciones de datos concurrentes. Las consultas y los datos que forman la base de datos tienen gran relevancia en toda la industria. Este benchmark ilustra sistemas de soporte de decisiones que examinan grandes volúmenes de datos, ejecutan consultas con un alto grado de complejidad y dan respuestas a preguntas críticas de negocio [115].

La base de datos TPC-H debe implementarse usando un DBMS comercialmente disponible y las consultas deben ejecutarse mediante una interfaz usando SQL dinámico. TPC-H no representa la actividad de ningún segmento de negocio particular, sino el de cualquier industria que debe gestionar, vender, o distribuir un producto en todo el mundo (por ejemplo, renta de autos, distribución de comida, partes, proveedores, etc.). Los componentes de la base de datos TPC-H consisten de ocho tablas individuales: **part, partsupp, customer,**

supplier, lineitem, orders, nation, region.

5.3.2. Implementación de los fragmentos verticales

Usamos la tabla **partsupp** del TPC-H de 1 GB, **partsupp** tiene 800,000 tuplas y 5 atributos. En la mayoría de los sistemas de bases de datos comerciales de hoy, no hay soporte DDL nativo para definir fragmentos verticales de una tabla [65]. Por lo tanto, estos sólo pueden implementarse con tablas, índices o vistas materializadas. Si los fragmentos se implementan como tablas, esto puede causar el problema de la elección del fragmento óptimo para una consulta. Por ejemplo, suponiendo que tenemos la tabla

partsupp

(*ps_partkey,*

ps_suppkey,

ps_availqty,

ps_supplycost,

ps_comment)

Fragmentos de **partsupp**:

partsupp_1(*ps_partkey, ps_psavailqty, ps_suppkey, ps_supplycost*)

partsupp_2(*ps_partkey, ps_comment*)

Donde *ps_partkey* es la clave primaria. Considerando una consulta:

`SELECT` *ps_partkey, ps_comment* `FROM` **partsupp**

La consulta de selección de **partsupp** no puede transformarse a la selección de **partsupp_2** por el optimizador de consultas automáticamente. Si los fragmentos se implementan como vistas materializadas, el procesador de consultas en el DBMS puede detectar la vista materializada óptima para una consulta y es capaz de reescribir la consulta para acceder a la vista materializada óptima. Si las particiones se implementan como índices sobre tablas relacionales, el procesador de consultas es capaz de detectar que el recorrido horizontal de un índice es equivalente a una búsqueda completa en un fragmento. Por lo tanto, implementar las particiones como vistas materializadas o como índices permite que

los cambios de los fragmentos sean transparentes para las aplicaciones [47].

5.3.3. Ilustración

DYVEP se implementa como un script SQL, el DBA que quiere fragmentar una tabla sólo tiene que ejecutar DYVEP.sql en la base de datos que contiene la tabla que desea fragmentar. DYVEP detectará que es la primera ejecución y creará las tablas, funciones y disparadores para implementar la fragmentación vertical dinámica.

Fragmentación vertical inicial (estática)

Paso 1. El primer paso de DYVEP es crear una fragmentación vertical inicial, para esto, el *Recolector de Estadísticas* de DYVEP analiza las consultas almacenadas en el registro de consultas y copia en la tabla **queries (QT)** las consultas ejecutadas en la tabla que se desea fragmentar.

Paso 2. Cuando el *Recolector de Estadísticas* copia todas las consultas, entonces se dispara el *Algoritmo de Fragmentación Vertical (VPA)*, DYVEP puede usar cualquier algoritmo que utilice como entrada la matriz de uso de atributos **(AUT)**, como ejemplo, en DYVEP implementamos el *VPA* de Navathe et al. [36], debido a que es un algoritmo clásico de fragmentación vertical.

Paso 3. El *Algoritmo de Fragmentación* primero obtendrá la **AUT** de la **QT**, la **AUT** tiene dos disparadores para cada atributo de la tabla que desea fragmentarse, uno para insertar y eliminar y otro para actualizar, en este caso tenemos los disparadores **inde_ps_partkey, update_ps_partkey**, etc., estos disparadores proveen la habilidad de actualizar la **attribute_ affinity_table (AAT)** cuando la frecuencia de los atributos usados por las consultas sufren cambios en la **AUT**. Cada vez que se inserta, elimina o modifica una consulta q_k en **QT**, se disparan las reglas **inde_a**$_i$ (la cual incluye las dos reglas **insert_a**$_i$ y **delete_a**$_i$), ..., **inde_a**$_n$ y **update_a**$_i$, ..., **update_a**$_n$. Estas reglas se definen de la siguiente forma:

insert_a_i

ON insert **AUT**

IF new.a_i=true

THEN for each $a_j \in$ **AT**, $i \leq j \leq n$ |**AUT**(q_k,a_j)=true

update **AAT**(a_i,a_j)=**AAT** (a_i,a_j)+new.*frequency*

update **AAT**(a_i,a_j)=**AAT**(a_i,a_j)+new.*frequency*

delete_a_i

ON delete **AUT**

IF old.a_i=true

THEN for each $a_j \in$ **AT**, $i \leq j \leq n$ | **AUT**(q_k,old.a_j)=true

update AAM(a_i,a_j)=**AAT**(a_i,a_j)-old.*frequency*

update AAM(a_j,a_i)=**AAT**(a_j,a_i)-old.*frequency*

update_a_i

ON update **AUT**

IF new.*frequency*<>old.*frequency*

THEN *freq*=new.*frequency*-old.*frequency*

for each $a_j \in$ **AT**, $i \leq j \leq k$ | **AUT**(q_k,a_j)=true

update **AAT**(a_i,a_j)=**AAT**(a_i,a_j)+*freq*

update **AAT**(a_j,a_i)=**AAT**(a_j,a_i)+*freq*

Paso 4. Cuando se actualiza la **AAT**, se dispara un procedimiento llamado BEA (*Bond Energy Algorithm*), el cual es un procedimiento general para permutar filas y columnas de una matriz cuadrada con el fin de obtener un semibloque en forma diagonal [36]. Este algoritmo se aplica generalmente para dividir un conjunto de variables de interacción en subconjuntos que interactúan mínimamente. La aplicación del procedimiento BEA a la **AAT** genera la **clustered_affinity_table (CAT)**.

Paso 5. Una vez que se genera la **CAT** se dispara un procedimiento llamado *partition()* que recibe como entrada la **CAT** y obtiene el Esquema de Fragmentación Vertical (EFV)

óptimo.

Paso 6. Cuando obtiene el EFV, VPA dispara el generador de fragmentos que materializa el EFV, esto es, crea los fragmentos en disco.

Los fragmentos se implementaron como índices debido a que el procesador de consultas de PostgreSQL puede detectar los índices y es capaz de reescribir la consulta para acceder a los índices en vez de las tablas completas, por lo tanto, los fragmentos son transparentes para los usuarios.

Una captura de pantalla de DYVEP se muestra en la Figura 5.2. Un esquema llamado DYVEP se crea en la base de datos TPC-H. En dicho esquema se ubican todas las tablas (**queries, attribute_usage_table, attribute_affinity_table, clustered_affinity_table**) del sistema DYVEP, los disparadores **inde_a$_i$, ..., inde_a$_n$** y **update_a$_i$, ..., update_a$_n$** se generan automáticamente por DYVEP de acuerdo a la vista *attributes*, por lo que el número de disparadores en nuestro sistema dependerá del número de atributos de la tabla que se desea fragmentar. El número de disparadores en DYVEP (#AR) puede obtenerse de acuerdo a la siguiente formula:

$$\#AR = 2 * n + 7 \tag{5.1}$$

donde n es el número de atributos de la tabla que se fragmenta verticalmente.

Teniendo las siguientes consultas

q$_1$:SELECT *ps_partkey, ps_availqty* FROM **partsupp** WHERE *ps_availqty*<Value

q$_2$:SELECT *ps_suppkey, ps_availqty* FROM **partsupp** WHERE *ps_availqty*=Value

q$_3$:SELECT *ps_partkey, ps_availqty, ps_comment* FROM **partsupp**
WHERE *ps_availqty*<Value

q$_4$:SELECT *ps_partkey, ps_suppkey, ps_supplycost* FROM **partsupp**
WHERE *ps_supplycost*>Value

q$_5$:SELECT *ps_partkey, ps_supplycost* FROM **partsupp**
WHERE **ps_supplycost**=Value

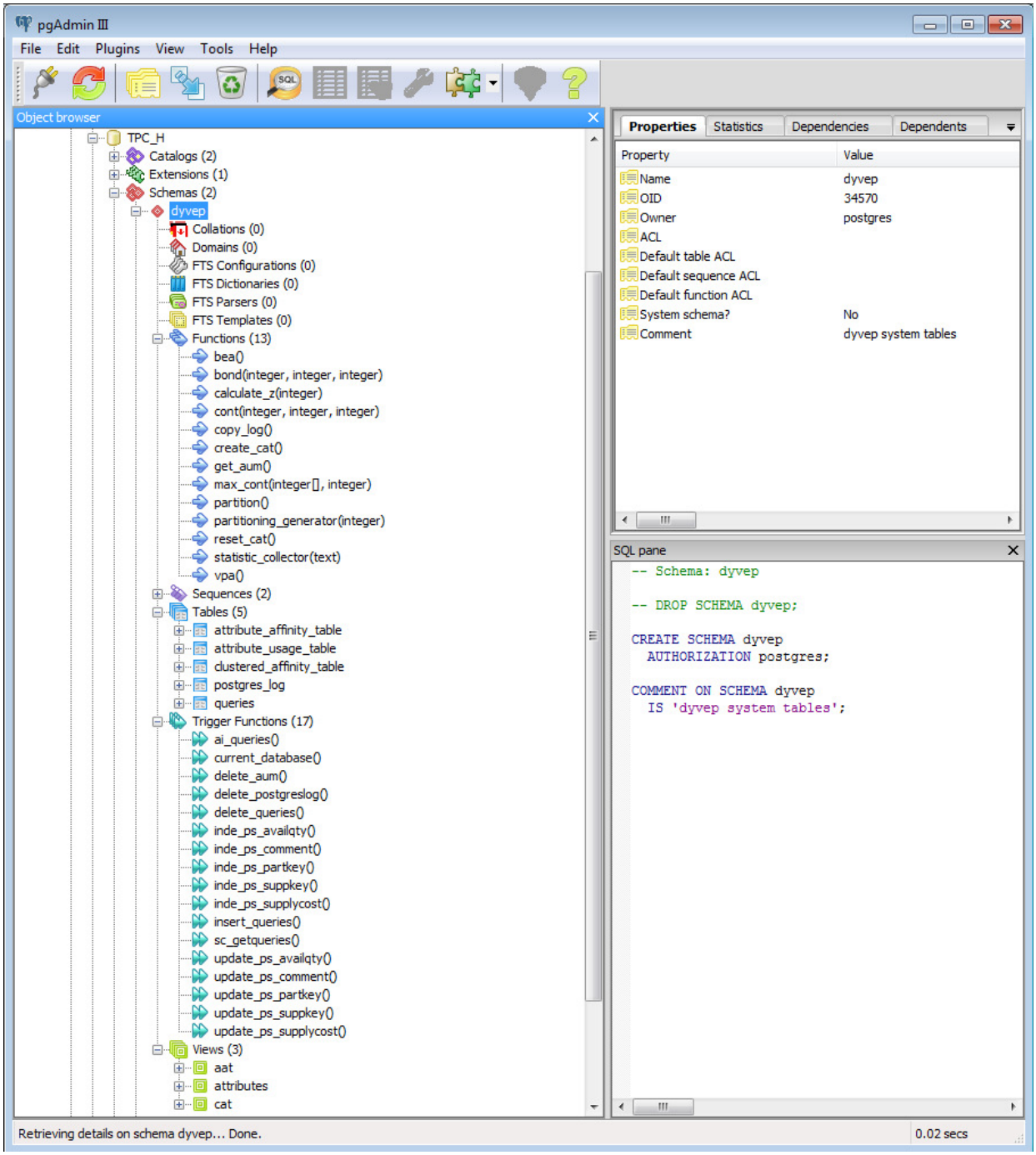
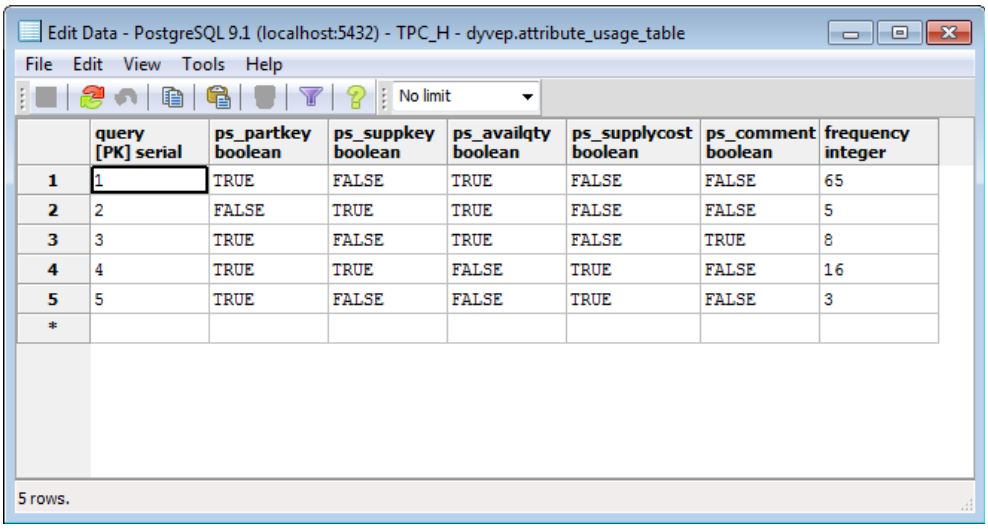


Figura 5.2: Captura de pantalla de DYVEP en PostgreSQL.



	query [PK] serial	ps_partkey boolean	ps_suppkey boolean	ps_availqty boolean	ps_supplycost boolean	ps_comment boolean	frequency integer
1	1	TRUE	FALSE	TRUE	FALSE	FALSE	65
2	2	FALSE	TRUE	TRUE	FALSE	FALSE	5
3	3	TRUE	FALSE	TRUE	FALSE	TRUE	8
4	4	TRUE	TRUE	FALSE	TRUE	FALSE	16
5	5	TRUE	FALSE	FALSE	TRUE	FALSE	3
*							

Figura 5.3: Tabla de uso de atributos.

Tabla 5.1: Comparación de tiempo de ejecución de consultas.

TPC-H	q_1	q_2	q_3	q_4	q_5
NF	3385 ms.	421 ms.	515 ms.	8674 ms.	531 ms.
DYVEP	344 ms.	78 ms.	172 ms.	8393 ms.	140 ms.

DYVEP obtuvo la tabla de uso de atributos de la Figura 5.3. El EFV generado por DYVEP de acuerdo a la tabla de uso de atributos fue:

partsupp_1(*ps_partkey, ps_psavailqty, ps_suppkey, ps_supplycost*)

partsupp_2(*ps_partkey, ps_comment*)

La Tabla 5.1 muestra el tiempo de ejecución de las consultas en la base de datos TPC-H no fragmentada (NF) y fragmentada verticalmente utilizando DYVEP. Como puede verse, el tiempo de ejecución de las consultas en TPC-H fragmentada verticalmente usando DYVEP es más bajo que en TPC-H no fragmentada, por lo tanto, DYVEP puede generar esquemas que mejoran significativamente la ejecución de las consultas.

Fragmentación vertical dinámica

Para realizar la fragmentación vertical dinámica es necesario que el DBA cree un regla activa que se ejecute cada determinado tiempo. Para crear este tipo de disparadores en PostgreSQL utilizamos PgAgent que es un programador de tareas que puede administrarse usando pgAdmin. La fragmentación vertical dinámica se realiza en DYVEP de acuerdo a los siguientes pasos:

Paso 1. El DBA debe ejecutar el archivo *dyvep_jobs.sql* en la base de datos `postgres`.

Paso 2. El script *dyvep_jobs.sql* crea el esquema *dyvep_jobs* en la base de datos `postgres` y dentro de este esquema crea la función *create_jobs(text)*.

Paso 3. El DBA ejecuta la función *create_jobs('database')* en la base de datos `postgres`, donde *'database'* es el nombre de la base de datos que contiene la tabla que se desea fragmentar.

Paso 4. La función *create_jobs('database')* crea el disparador temporal (job) en postgres llamado **sc_dyvep** que elimina todas las consultas que se encuentran almacenadas en la tabla **postgres_log**, el job **sc_dyvep** se ejecutará todos los días a las 23:58 porque a esa hora el log de transacciones contendrá todas las consultas que se realizaron durante el día en la tabla que se desea fragmentar. La descripción de este disparador es:

sc_dyvep

`ON` 23:58

`THEN` delete from **postgres_log**

Paso 5. Cuando las consultas del día anterior se eliminan en la tabla **postgres_log** se copian las consultas del día actual en la tabla **postgres_log**, de esto se encarga el disparador **delete_postgreslog** que se ejecuta después de eliminar en la tabla **postgres_log**.

delete_postgreslog

`ON` delete from **postgres_log**

`THEN` call *copy_log()*

Paso 6. Antes de insertar en la tabla **postgres_log,** se dispara **current_database** que verifica que sólo se inserten las consultas de la base de datos actual.

current_database

`ON` before insert **postgres_log**

`IF` new.*database_name*=current_database() and new.*day_execution*=current_date()

`THEN` return new

Paso 7. Después de insertar en la tabla **postgres_log,** el disparador **sc_getqueries** verifica si existe la vista *attributes*, en tal caso ejecuta *statistic_collector('tabla')*, el nombre de la tabla se obtiene de la vista *attributes*.

sc_getqueries

`ON` after insert **postgres_log**

`IF` *dyvep.attributes* ∈ **pg_views**

`THEN` select into *t_name* "*table*" from *dyvep.attributes* where *att_order*=1;

execute 'select *dyvep.statistic_collector('t_name')'*

Paso 8. La función *statistic_collector('tabla')* debe detectar que no es la primera ejecución de DYVEP, obtener las nuevas estadísticas y disparar la función *vpa()*.

Paso 9. La función *vpa()* obtiene las nuevas **attribute_usage_table**, **attribute_affinity_table**, **clustered_affinity_table**, genera el nuevo EFV y dispara la función *partitioning_generator(integer)*.

Paso 10. Finalmente, la función *partitioning_generator(integer)* elimina los fragmentos anteriores y crea los nuevos.

DYVEP puede adaptar el EFV cuando se realizan los siguientes cambios en las consultas almacenadas en la tabla **queries**:

Eliminar consultas.

En DYVEP nunca se eliminan consultas de la tabla **queries** porque no tendría caso eliminarlas, ya que en la fragmentación vertical una consulta se considera igual si usa los mismos atributos. Por ejemplo, las consultas:

SELECT SUM(*ps_availqty*) FROM **partsupp** WHERE *ps_partkey*=1;

SELECT *ps_partkey*, *ps_availqty* FROM **partsupp** WHERE *ps_availqty*<10;

Son iguales, debido a que las dos acceden a los mismos atributos, si se tiene un fragmento que contenga los dos atributos *ps_availqty* y *ps_partkey* entonces las dos consultas sólo accederán a dicho fragmento sin acceder a atributos irrelevantes y accediendo a todos los atributos relevantes que son las dos metas que se pretenden lograr al fragmentar verticalmente una tabla. Aunque la primera consulta recupera sólo 1 fila y la segunda recupera 741 filas, no nos interesa reducir las tuplas irrelevantes accedidas por las consultas porque en ese caso se estaría hablando de una fragmentación híbrida, la cual no esta dentro de esta investigación. En el futuro se piensa desarrollar métodos híbridos de fragmentación para bases de datos relacionales.

No tiene caso eliminar consultas porque la tabla queries solo almacena patrones de consultas, entonces nunca almacenará dos consultas que utilicen los mismos atributos, todas las consultas almacenadas en queries acceden a diferentes conjuntos de atributos. Es decir, si se ejecuta la primera consulta por primera vez en DYVEP ésta se almacenará en **queries** pero si posteriormente se ejecuta la segunda consulta, ésta no se almacenará porque usa los mismos atributos, por lo que sólo se incrementará su frecuencia.

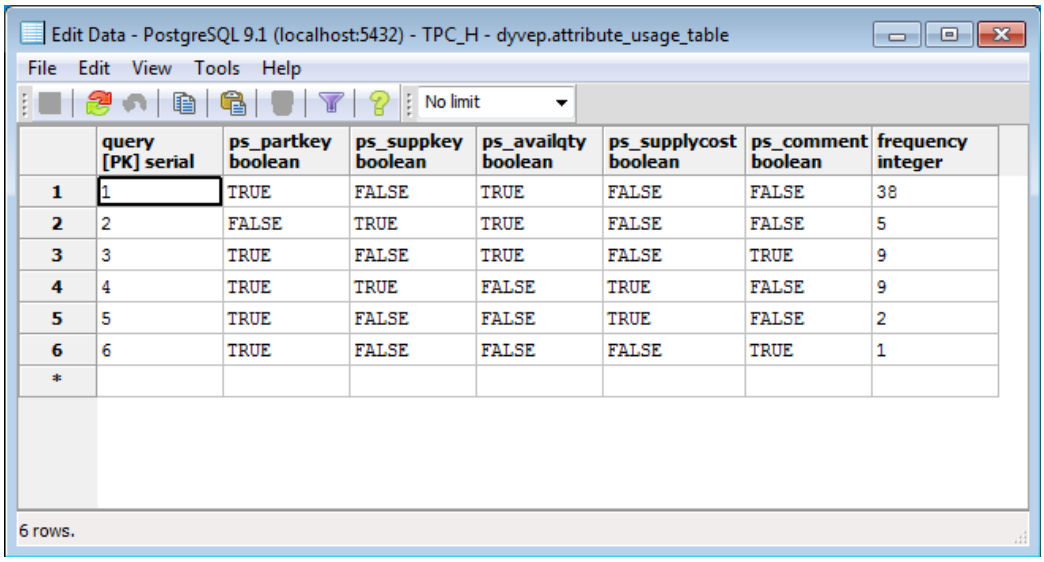
Además si en un día no se ejecuta ninguna consulta almacenada en la tabla queries, su frecuencia será igual a cero por lo que no afectará el resultado de la fragmentación vertical.

Agregar consultas

Así que el único cambio que puede ocurrir en DYVEP en cuanto al número de consultas es que se agreguen nuevas consultas. En la Figura 5.4 la tabla de uso de atributos obtenida por DYVEP muestra que se realizó una nueva consulta.

q_6: SELECT *ps_partkey*, *ps_comment* FROM **partsupp**

WHERE *ps_comment*=Value



	query [PK] serial	ps_partkey boolean	ps_suppkey boolean	ps_availqty boolean	ps_supplycost boolean	ps_comment boolean	frequency integer
1	1	TRUE	FALSE	TRUE	FALSE	FALSE	38
2	2	FALSE	TRUE	TRUE	FALSE	FALSE	5
3	3	TRUE	FALSE	TRUE	FALSE	TRUE	9
4	4	TRUE	TRUE	FALSE	TRUE	FALSE	9
5	5	TRUE	FALSE	FALSE	TRUE	FALSE	2
6	6	TRUE	FALSE	FALSE	FALSE	TRUE	1
*							

Figura 5.4: Tabla de uso de atributos 2.

Cambiar la frecuencia de las consultas

En caso de que ocurran cambios en la frecuencia de las consultas, DYVEP los detectará y generará el nuevo EFV. Por ejemplo, en la Figura 5.4 se presenta la tabla de uso de atributos generada por DYVEP, en la cual puede verse que cambió la frecuencia con que se realizaron las consultas.

Estos cambios generaron un nuevo EFV:

partsupp_1(*ps_partkey, ps_supplycost*)

partsupp_2(*ps_partkey, ps_psavailqty, ps_suppkey, ps_comment*)

En la Tabla 5.2 se presenta el tiempo de ejecución de las consultas en la base de datos TPC-H no fragmentada (NF), en el EFV inicial generado por DYVEP (fragmentación estática, FE) y en el nuevo EFV obtenido por DYVEP (fragmentación dinámica, FD). Como puede verse, el tiempo de ejecución de las consultas es más bajo en TPC-H fragmentada verticalmente usando DYVEP (tanto en la fragmenación estática como en la dinámica) que en TPC-H no fragmentada, por lo tanto, DYVEP puede generar esquemas que reducen significativamente el tiempo de ejecución de las consultas, aún cuando cambien los patrones

Tabla 5.2: Comparación de tiempo de ejecución de consultas.

TPC-H	q_1	q_2	q_3	q_4	q_5	q_6
NF	3385 ms.	421 ms.	515 ms.	8674 ms.	531 ms.	13026 ms.
DYVEP FE	344 ms.	78 ms.	172 ms.	8393 ms.	140 ms.	1389 ms.
DYVEP FD	203 ms.	218 ms.	218 ms.	8268 ms.	109 ms.	234 ms.

de acceso a la base de datos.

5.4. Discusión

Las principales ventajas de DYVEP sobre otras propuestas son:

1. Las estrategias de fragmentación vertical estática realizan una fase de análisis de la base de datos para recolectar la información necesaria en el proceso de fragmentación vertical, además en algunas herramientas automáticas de fragmentación vertical [67,83] es necesario que el DBA dé la carga de trabajo como entrada al proceso de fragmentación. En contraste, DYVEP implementa un *Recolector de Estadísticas* basado en reglas activas que acumula información acerca de los atributos y consultas sin la explícita intervención del DBA.

2. Cuando la información de las consultas cambia en las estrategias de fragmentación vertical estáticas, el EFV seguirá igual y no se implementará la mejor solución. En DYVEP, el EFV cambia dinámicamente de acuerdo a los cambios en los patrones de acceso a la base de datos con el fin de encontrar la mejor solución y no afectar el desempeño de la base de datos.

3. El proceso de fragmentación vertical en las propuestas estáticas se ejecuta fuera de la base de datos y cuando se encuentra el EFV se materializan los fragmentos verticales. En DYVEP todo el proceso de fragmentación vertical se implementa dentro de la base de datos usando reglas activas, la matriz de uso de atributos (AUM) utilizada por la mayoría de los algoritmos de fragmentación vertical se implementa como una tabla de la base de datos (**AUT**) con el fin de usar reglas o disparadores para cambiar el

EFV automáticamente.

4. Algunas herramientas de fragmentación vertical [83] sólo recomiendan el EFV óptimo pero dejan la creación de los fragmentos al DBA, DYVEP tiene un *Generador de Fragmentos* que crea automáticamente los fragmentos en disco.

5.5. Conclusión

Se diseñó e implementó la arquitectura de un sistema activo para fragmentación vertical dinámica de bases de datos relacionales llamado DYVEP. Usando reglas activas DYVEP puede modificar el EFV de una base de datos de acuerdo a los cambios en sus patrones de acceso con el objetivo de mejorar el tiempo de respuesta de las consultas. En el siguiente Capítulo se extiende nuestro sistema a bases de datos multimedia porque este tipo de bases de datos son altamente dinámicas, por lo tanto, DYVEP reduce considerablemente el tiempo de respuesta de las consultas en estas bases de datos.

Capítulo 6

DYMOND: Un Sistema Activo de Fragmentación Vertical Dinámica para Bases de Datos Multimedia

En el Capítulo 4 se explicó la importancia de utilizar técnicas de fragmentación vertical en bases de datos multimedia para lograr la recuperación eficiente de los objetos multimedia en estas bases de datos. Por lo tanto, en el mismo Capítulo se propuso un algoritmo de fragmentación vertical para bases de datos multimedia que toma en cuenta el tamaño de los atributos (MAVP). En dicho algoritmo consideramos un conjunto de consultas y sus frecuencias, así como los tamaños de los atributos como entrada al proceso de fragmentación vertical, MAVP obtiene un EFV de acuerdo a dicha información. MAVP sólo obtiene el EFV pero es el DBA quien debe realizar una fase de análisis previa para recolectar la información que MAVP utiliza como entrada, ejecutar MAVP para obtener un EFV y materializar el EFV, es decir, crear los fragmentos en disco.

Las bases de datos multimedia son accedidas por muchos usuarios simultáneamente [23], por lo tanto, las consultas y sus frecuencias tienden a cambiar rápidamente con el tiempo, en este contexto, el DBA tendría que realizar el análisis de la base de datos, ejecutar MAVP

y materializar el EFV cada vez que los patrones de acceso sufran suficientes cambios para no degradar el desempeño de la base de datos. Esto implica que el DBA debe saber en qué momento iniciar un nuevo proceso de fragmentación. Por lo tanto, es necesario un mecanismo para saber qué cambios se consideran suficientes para disparar un nuevo proceso de fragmentación.

La fragmentación vertical dinámica ahorra el costo de procesamiento eliminando la necesidad de realizar la fase de análisis previa antes de cada nueva fragmentación, también mejora el desempeño total del sistema porque es capaz de detectar oportunamente cuando el desempeño de la base de datos cae bajo un umbral. Por estas razones, es más adecuado desarrollar un sistema de fragmentación vertical dinámica para bases de datos multimedia inteligente que sea capaz de ejecutarse por sí mismo cuando sea más apropiado.

En el Capítulo anterior describimos nuestra propuesta de un sistema activo para fragmentación vertical dinámica en bases de datos relacionales, en este Capítulo extendemos dicho sistema para utilizarlo en bases de datos multimedia y le llamamos DYMOND (DYnamic Multimedia ON line Distribution).

6.1. Motivación

A continuación se presentan las desventajas de DYVEP, las cuales se resuelven en DYMOND:

6.1.1. Desventajas de DYVEP

Una desventaja de DYVEP es que realiza todo el proceso de fragmentación en cada ejecución. Es decir, cada vez que se ejecuta DYVEP (por ejemplo, cada día) recolecta estadísticas, obtiene un nuevo EFV y materializa el EFV, aunque la tabla contenga las mismas consultas con la misma frecuencia, como puede verse en el Algoritmo 6.1 .

Otra desventaja de DYVEP es que no considera los cambios en los atributos, por lo que si se eliminan, renombran o agregan atributos a la tabla fragmentada dinámicamente, el DBA debe eliminar el esquema *dyvep* de la base de datos y volver a crearlo para que

Algoritmo 6.1 DYVEP

Entrada:
DB Base de datos
log SQL Registro de consultas
Salida:
DB Base de datos
begin
if first_execution **then**
    create EFV inicial
else
    Recolectar estadísticas
    Obtener nuevo EFV
    Materializar EFV
end if
end. {DYVEP}

se vuelvan a generar las tablas **attribute_usage_table**, **attribute_affinity_table**, **clustered_affinity_table** y las vistas *aat* y *cat*.

En DYMOND se eliminan dichos problemas, ya que sólo se creará un nuevo EFV cuando los patrones de acceso a la base de datos hayan sufrido cambios mayores a los cambios promedio y cuando el costo del EFV actualmente implementado sea mayor a un umbral previamente determinado por DYMOND, como se muestra en el Algoritmo 6.2. Además, DYMOND si toma en cuenta el cambio en los atributos.

6.1.2. Objetivo de DYMOND

DYMOND tiene como objetivo principal permitir la recuperación eficiente de objetos multimedia de la base de datos en todo tiempo. DYMOND utiliza reglas activas para monitorizar consultas con el fin de recolectar toda la información necesaria por MAVP, analizar la información para determinar si es necesaria una nueva fragmentación, y en tal caso disparar el algoritmo MAVP. En DYMOND adoptamos la técnica de fijar un umbral de desempeño de la MMDB distribuida después de cada fragmentación y una variable determinada por el sistema que mide el porcentaje de cambio de los datos de entrada a MAVP a través del tiempo [59]. DYMOND monitoriza continuamente los cambios en el

Algoritmo 6.2 DYMOND

```
Entrada:
MMDB Base de datos multimedia
log_SQL Registro de consultas
Salida:
MMDB Base de datos multimedia
begin
if first_execution then
    create EFV inicial
else
    Recolectar estadísticas
    if cambios>cambios_promedio then
        Obtener nuevo costo EFV
        if nuevo_costo_EFV>threshold then
            Obtener nuevo EFV
            Determinar threshold
            Materializar EFV
        end if
    end if
end if
end. {DYMOND}
```

esquema de la base de datos, así como en la información de los patrones de acceso usando reglas activas y utiliza dicha información para evaluar el desempeño del sistema basado en el EFV actual. El desempeño del sistema se mide en términos de:

1. La cantidad de accesos locales irrelevantes, que producen una medida del costo de procesamiento local de las consultas debido a atributos irrelevantes (IAAC).

2. La cantidad de accesos remotos relevantes debido a atributos relevantes de fragmentos que las consultas acceden remotamente (TC).

DYMOND utiliza esta medida para determinar un umbral de desempeño de la MMDB y cuándo disparar una nueva fragmentación.

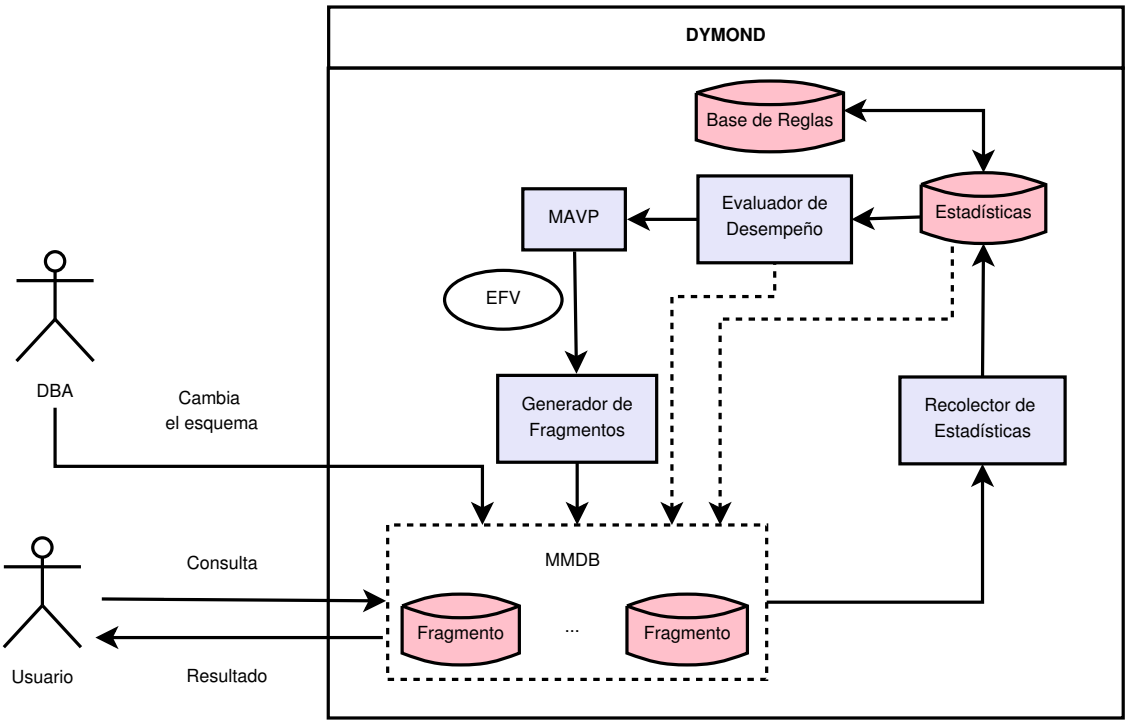


Figura 6.1: Arquitectura de DYMOND.

6.2. Arquitectura de DYMOND

La arquitectura de DYMOND se muestra en la Figura 6.1, ésta se compone de cuatro módulos: *Recolector de Estadísticas, Evaluador de Desempeño, MAVP y Generador de Fragmentos*. Cuando un usuario envía una consulta a la MMDB o cuando el DBA realiza cambios en el esquema de la base de datos, el *Recolector de Estadísticas* de DYMOND evalúa la información de las consultas y los atributos y compara la nueva información con la anterior con el fin de determinar si los cambios son suficientes para disparar el *Evaluador de Desempeño*. Cuando se dispara el *Evaluador de Desempeño*, el costo del EFV se calcula, si su costo es mayor que el umbral de desempeño entonces se dispara MAVP, que obtiene un nuevo EFV. Finalmente, el *Generador de Fragmentos* materializa el nuevo EFV.

6.2.1. Recolector de estadísticas

El *Recolector de Estadísticas* acumula toda la información que MAVP necesita para obtener un EFV, dicha información son los atributos utilizados por las consultas (A), la frecuencia de las consultas (F), el número de consultas (NQ) el tamaño de los atributos (S) y el número de atributos (NA). Además, registra todos los cambios en los datos de entrada a través del tiempo y compara los nuevos cambios con los cambios anteriores para determinar si son suficientes para disparar el *Evaluador de Desempeño*. Sólo se evaluará el desempeño de la MMDB si durante la verificación de cambio se encuentra un cambio de entrada que es mayor que el cambio promedio del sistema.

Un cambio en cualquier dato de entrada NA, S, NQ, F se calcula como el porcentaje de los datos originales. El cambio en el número de atributos (*currentChangeA*) en la MMDB se determina por el porcentaje del número de atributos insertados o eliminados después de la última fragmentación (*currentNA*) sobre el número de atributos que tenía la tabla antes de la última fragmentación (*previousNA)*, esto se define como:

$$currentChangeA = \frac{currentNA}{previousNA} * 100 \tag{6.1}$$

Esto significa que en una MMDB con 8 atributos, si se agrega o se elimina un atributo, esto se define como $\frac{1}{8} * 100 = 12{,}5\,\%$ de cambio.

El cambio en el tamaño de los atributos (*currentChangeS*) se define como el máximo porcentaje de cambio en cualquier atributo. Por lo tanto, este valor se calcula como el máximo del total de cambio dividido por el tamaño que tenía el atributo en la última fragmentación (*prev_ size*) multiplicado por 100.

$$currentChangeS = \frac{size - prev\_size}{prev\_size} * 100 \tag{6.2}$$

Para obtener estos valores, en DYMOND existen las siguientes tablas:

attribute (*id, attribute, prev_ size, size*) almacena el identificador, el nombre, el tamaño de los atributos de la última fragmentación y el tamaño de los atributos después de la última

fragmentación.

stat_attributes (*currentNA, previousNA, currentChangeA, previousChangeA, currentChangeS, previousChangeS*). A continuación se definen los atributos de esta tabla:

- *currentNA.* Número de atributos insertados o eliminados desde la última fragmentación.
- *previousNA.* Número de atributos que había en la última fragmentación, estos se encuentran en **attribute.**
- *currentChangeA.* Porcentaje de cambio de los atributos.
- *previousChangeA.* Porcentaje de cambio de los atributos de la última fragmentación.
- *currentChangeS.* Porcentaje de cambio del tamaño de los atributos.
- *previousChangeS.* Porcentaje de cambio del tamaño de los atributos de la última fragmentación.

stat_flags (*flag_ att, flag_ queries, flag_frequency, flag_ ca, flag_ cs, flag_ cq, flag_ cf*)

- *flag_ att.* Variable que indica si ya se evaluaron los cambios en todos los atributos.
- *flag_ queries.* Es verdera cuando se analizan los cambios en todas las consultas.
- *flag_frequency.* Indica el momento en que se evalúan los cambios en la frecuencia de todas las consultas
- *flag_ ca.* Es verdadera si ocurren cambios en los atributos mayores a los cambios promedio.
- *flag_ cs.* Bandera de cambio en el tamaño de los atributos mayores a los cambios promedio.
- *flag_ cq.* Variable que indica si ocurren cambios en el número de consultas mayores al cambio promedio.

- *flag_ cf.* Bandera de cambio en la frecuencia de las consultas mayores al cambio promedio.

El *Recolector de Estadísticas* implementa las siguientes reglas en las tablas anteriores:

Regla 1 (id_att)

`ON` insert or delete **attribute**

`THEN` update **stat_attributes** set *currentNA*=*currentNA*+1

Regla 2 (update_currentna)

`ON` update **stat_attributes**.*currentNA*

`IF` *currentNA*>*0* and *previousNA*>0

`THEN` update **stat_attributes** set $currentChangeA = \frac{currentNA}{previousNA} * 100$

Regla 3 (update_currentchangea)

`ON` update **stat_attributes**.*currentChangeA*

`IF` *currentChangeA*>*previousChangeA*

`THEN` update **stat_flags** set *flag_ ca*=true

Regla 4 (update_att)

`ON` update **atribute**.*size*

`IF` *prev_ size*>0

`THEN` update **stat_attributes** set

$$currentChangeS = \frac{\textbf{attribute}.size - \textbf{attribute}.prev\_size}{\textbf{attribute}.prev\_size} * 100$$

Regla 5 (update_currentchanges)

`ON` before update **stat_attributes**.*currentChangeS*

`IF` *currentChangeS*<0

`THEN` update **stat_attributes** set *currentChangeS*=*currentChangeS**-1

Regla 6 (update_currentchanges)

`IF` old.*currentChangeS*>*currentChangeS*

`THEN` update **stat_attributes** set *currentChangeS*=old.*currentChangeS*

Regla 7 (after_update_currentchanges)

`ON` update **stat_attributes**.*currentChangeS*

`IF` *currentChangeS>previousChangeS*

`THEN` update **stat_flags** set *flag_cs*=true

Ejemplo 6.1 *Si inmediatamente después de una fragmentación inicial tenemos la MMDB con la relación* ***EQUIPO****(id, nombre, imagen, gráfico, audio, video), previousNA=6 (porque* ***EQUIPO*** *tiene 6 atributos) y previousChangeA=0 (porque después de una fragmentación inicial todos los cambios previos son iguales a 0), y se agrega un nuevo atributo, el Recolector de Estadísticas dispara la* ***Regla 1,*** *por lo tanto, actualiza currentNA=1 en* ***stat_attributes****, esta operación genera la detección del evento update* ***stat_attributes****.currentNA, el cual dispara la* ***Regla 2*** *que actualiza currentChangeA* $= \frac{1}{6} * 100 = 16{,}6\,\%$*, esta acción trae como consecuencia la detección del evento update* ***stat_attributes.****currentChangeA, el cual dispara la* ***Regla 3****, la evaluación de su condición es verdadera, ya que currentChangeA (16.6) es mayor a previousChangeA (0), así que el Recolector de Estadísticas actualiza flag_ca=true en* ***stat_flags****.*

El cambio en las consultas que acceden a la MMDB *(currentChangeQ)* se calcula como el porcentaje del número de nuevas consultas que la acceden (*currentNQ*) sobre el número de consultas que accedían a la MMDB antes de la última fragmentación (*previousNQ*). Esto puede expresarse como:

$$currentChangeQ = \frac{currentNQ}{previousNQ} * 100 \tag{6.3}$$

El cambio en la frecuencia de acceso de las consultas (*currentChangeF*) se define como el máximo porcentaje de cambio en cualquier consulta. Por lo tanto, el cambio en la frecuencia de acceso de las consultas se calcula como el máximo del total de cambio dividido por la frecuencia para esa consulta (*prev_frequency*) antes de la última fragmentación, multiplicado por 100.

$$currentChangeF = \frac{frequency - prev\_frequency}{prev\_frequency} * 100 \tag{6.4}$$

Para obtener estos valores el *Recolector de Estadísticas* utiliza las siguientes tablas:

queries(*id, description, prev_frequency, frequency*) que almacena un identificador de las consultas que se realizan sobre la tabla que se va a fragmentar, su descripción, su frecuencia de la última fragmentación y su frecuencia después de la última fragmentación.

stat_queries (*currentNQ*, *previousNQ*, *currentChangeQ*, *previousChangeQ*, *currentChangeF*, *previousChangeF*). A continuación se definen los atributos de esta tabla:

- *currentNQ.* Número de consultas insertadas desde la última fragmentación.
- *previousNQ.* Número de consultas que había en la última fragmentación.
- *currentChangeQ.* Porcentaje de cambio de las consultas.
- *previousChangeQ.* Porcentaje de cambio de las consultas de la última fragmentación.
- *currentChangeF.* Porcentaje de cambio de la frecuencia de las consultas.
- *previousChangeF.* Porcentaje de cambio de la frecuencia de las consultas de la última fragmentación.

El *Recolector de Estadísticas* implementa las siguientes reglas en las tablas anteriores:

Regla 8 (after_id_queries)

`ON` insert or delete **queries**

`THEN` update **stat_queries** set $currentNQ$=$currentNQ$+1

Regla 9 (update_currentnq)

`ON` update **stat_queries**.$currentNQ$

`IF` $currentNQ$>0 and $previousNQ$>0

`THEN` update **stat_queries** set $currentChangeQ = \frac{currentNQ}{previousNQ} * 100$

Regla 10 (update_currentchangeq)

`ON` update **stat_queries**.$currentChangeQ$

`IF` $currentChangeQ$>$previousChangeQ$

`THEN` update **stat_flags** set *flag_cq*=true

Regla 11 (delete_postgreslog)

ON update **queries**.*frequency*

IF *prev_frequency*>0 and *frequency*>0

THEN update **stat_queries** set

$$currentChangeF = \frac{\textbf{queries}.frequency - \textbf{queries}.prev\_frequency}{\textbf{queries}.prev\_frequency} * 100$$

Regla 12 (update_currentchangef)

ON before update **stat_queries**.*currentChangeF*

IF *currentChangeF*<0

THEN update **stat_queries** set *currentChangeF*=*currentChangeF**-1

Regla 13 (update_currentchangef)

ON before update **stat_queries**.*currentChangeF*

IF OLD.*currentChangeF*>*currentChangeF*

THEN update **stat_queries** set *currentChangeF*=OLD.*currentChangeF*

Regla 14 (after_update_currentchangef))

ON after update **stat_queries**.*currentChangeF*

IF *currentChangeF*>*previousChangeF*

THEN update **stat_flags** set *flag_cf*=true

El *Recolector de Estadísticas* dispara el *Evaluador de Desempeño* cuando al menos una de las variables de cambio almacenadas en **stat_flags** es verdadera y cuando ya analizaron todas las consultas y todos los atributos. Para esto se utiliza la regla activa:

Regla 15 (update_flag)

ON after update **stat_flags**

IF (*flag_att*=true AND *flag_queries*=true AND *flag_frequency*=true)

AND (*flag_ca*=true OR *flag_cs*=true OR *flag_cq*=true OR *flag_cf*=true)

THEN call Performance Evaluator.

6.2.2. Evaluador de desempeño

El *Evaluador de Desempeño* utiliza el modelo de costo presentado en el Capítulo 4 para obtener el nuevo costo del EFV actualmente implementado, ya que dicho modelo permite medir el costo de distintos EFVs en MMDBs. Si el *Evaluador de Desempeño* es disparado por el *Recolector de Estadísticas,* entonces debe calcular el costo de la ejecución de las consultas en el EFV actual, para esto obtiene la tabla de uso de atributos con el tamaño de los atributos, de acuerdo a esta información calcula el nuevo costo del EFV y dispara MAVP si el nuevo costo del EFV es mayor que un umbral de desempeño. Para esto utiliza la tabla:

stat_performance(*cost_VPS, performance_threshold, best_cost_VPS, step, sum, number, flag, s*). A continuación se define cada atributo:

- *cost_VPS.* Costo del EFV actual.
- *performance_threshold.* Umbral de desempeño de la MMDB se calcula

  *performance_threshold =best_cost_VPS*s.*
- *best_cost_VPS.* Costo del mejor EFV (EFV con menor costo obtenido por MAVP).
- *step.* Paso del árbol de partición que contiene el EFV con el menor costo.
- *sum.* Suma de los porcentajes de cambio en todos los datos de entrada de MAVP mayores a cero.
- *number.* Número de datos de entrada con porcentaje de cambio mayor a cero.
- *flag.* Variable que indica el momento en el que se obtienen todos los porcentajes de cambio de los datos de entrada.
- *s.* Variable de cambio de los datos de entrada a MAVP, se obtiene

  $s = 1 + \frac{10}{sum/number}$.

Regla 16 (performance_evaluator)

ON call Performance_Evaluator

THEN get_AUT

get cost(EFV_{actual})=IAAC(EFV_{actual})+TC(EFV_{actual})

update **stat_performance** set *cost_VPS*=cost(EFV_{actual});

Regla 17 (update_cost_vps)

ON update **stat_performance**.*cost_VPS*

IF *cost_VPS*>*performance_threshold*

THEN call MAVP

6.2.3. MAVP

MAVP obtiene un nuevo EFV usando la tabla de uso de atributos y actualiza el costo del nuevo EFV en **stat_performance.**

Regla 18 (mavp)

ON call MAVP

THEN MAVP(AUT, *EFVnew)*

get cost*(EFVnew)*=IAAC*(EFVnew)*+TC*(EFVnew)*

update **stat_performance** set *best_cost_VPS*=cost*(EFVnew)*

Después de obtener un nuevo EFV, MAVP calcula el umbral de desempeño (*performance_threshold*) para eso utiliza el costo del nuevo EFV (*best_cost_VPS*) almacenado en las estadísticas y la variable de cambio de la MMDB (s). El valor del umbral se calcula como:

$$performance\_threshold = best\_cost\_VPS * s. \tag{6.5}$$

La variable de cambio s mide o representa la frecuencia con la que cambia la información de entrada a MAVP y que puede afectar el desempeño, s se calcula como la inversa del porcentaje de cambio promedio en la entrada de *MAVP*, multiplicada por 10 para permitir

un margen razonable de degradación de desempeño más 1. El valor de s siempre es mayor que 1 para permitir que el umbral siempre sea mayor que el costo del nuevo EFV. Un umbral menor que el costo del nuevo EFV siempre disparará una nueva fragmentación, lo cual no es deseable.

Se usa la inversa porque una MMDB que cambia frecuentemente requiere una nueva fragmentación con mayor frecuencia y, por lo tanto, un valor de s más bajo, ya que mientras más bajo sea el valor de s, el umbral estará mas cerca del costo del nuevo EFV(mejor valor de desempeño). Mientras más bajo sea el umbral es más probable que el *Evaluador de Desempeño* detecte que el desempeño de la MMDB es más alto que el umbral y dispare *MAVP*. La implicación es que si la información de entrada a *MAVP* cambia frecuentemente, entonces su cambio promedio es muy alto haciendo su inversa muy baja o cercana a cero. Entonces el umbral de desempeño es cercano al costo del nuevo EFV. Mientras los patrones de acceso a la MMDB cambien con más frecuencia, menor será el valor s, mientras que un valor grande de s significa un porcentaje de cambio promedio muy bajo en los patrones de acceso.

MAVP es el encargado de calcular los cambios promedio y después obtener el umbral de desempeño. Una vez que se calculan los cambios promedio en los datos de entrada, la variable s se calcula simplemente dividiendo 10 entre el promedio de los porcentajes de cambio en todos los datos de entrada. Por ejemplo, una MMDB con un 100 % de cambio en el número de consultas (NQ), 80 % de cambio en la frecuencia de las consultas (F), 60 % de cambio en el tamaño de los atributos (S), y 50 % de cambios en el número de atributos (NA), el valor s se calcula como $s = 1 + \frac{10}{(100+80+60+50)/4}$=1.13, por otro lado, si el porcentaje de cambio promedio es 50 %, s se calcula como $s = 1 + \frac{10}{50}$=1.2. Esto muestra que un mayor cambio produce un valor menor de s, mientras que los porcentajes de cambio promedio bajos producen un valor de s más alto. El valor del umbral ahora se determina multiplicando *best_ cost_ VPS* por s. Esto significa que cualquier cambio menor o igual al cambio promedio no disparará una nueva fragmentación.

Después del proceso de fragmentación, *MAVP* debe calcular los promedios de cambio

en los datos de entrada para determinar el valor de la variable de cambio s y obtener el valor del umbral de desempeño. Con el fin de calcular el promedio de los cambios en los datos de entrada, MAVP necesita recordar los cambios previos en cada dato de entrada, así que debe calcular el promedio de los cambios encontrando la suma del cambio previo y el cambio actual. Posteriormente debe almacenar en las estadísticas los cambios promedio en los datos de entrada NQ, F, S, NA y actualizar el valor del cambio actual a cero. Para esto utiliza la siguiente regla:

Regla 19 (update_best_cost_vps)

`ON` update **stat_performance**.*best_cost_VPS*

`THEN` update **stat_attributes** set

previousNA=*currentNA*

currentNA=0

previousChangeA=(*previousChangeA*+*currentChangeA*)/2

currentChangeA=0

previousChangeS=(*previousChangeS*+*currentChangeS*)/2

currentChangeS=0

update **stat_queries** set

previousNQ=*currentNQ*

currentNQ=0

previousChangeQ=(*previousChangeQ*+*currentChangeQ*)/2

currentChangeQ=0

previousChangeF=(*previousChangeF*+*currentChangeF*)/2

currentChangeF=0

update **queries** set *prev_F*=*F*, *F*=0

update **attributes** set *prev_S*=*S*

update **stat_performance** set *flag*=true

Finalmente, para obtener el valor de la variable de cambio s, el umbral de desempeño y disparar el *Generador de Fragmentos, MAVP* utiliza la siguiente regla:

Regla 20 (update_flag_pt)

ON update **stat_performance.***flag*

IF *flag=true*

THEN update **stat_performance** set

sum=previousChangeA+previousChangeS+previousChangeQ+previousChangeF

number=values of changes >0

s=1+50/(*sum*/*number*)

*performance_threshold=best_cost_VPS*s*

call Partitioning_Generator

6.2.4. Generador de Fragmentos

El *Generador de Fragmentos* se encarga de materializar el nuevo EFV, para esto debe eliminar los fragmentos anteriores y crear los nuevos. Este implementa los fragmentos como índices, de esta forma los índices se usan automáticamente por el DBMS cuando se ejecuta una consulta y no es necesario cambiar la definición de las consultas para usar los fragmentos en vez de las tablas completas. Así que el uso de los fragmentos es transparente para los usuarios.

Regla 21 (partitioning_generator)

ON call partitioning_generator

IF **stat_performance**.*best_cost_vps*<**stat_performance**.*cost_vps*

THEN drop indexes

create indexes

fr_1=$index_1$, ..., fr_p=$index_p$

Este ejemplo muestra la ejecución de las reglas de DYMOND en la MMDB del ejemplo 6.1. Usando la AUM de la Tabla 6.1 MAVP obtiene el esquema *(id, audio, video), (nombre, imagen, gráfico)* con un costo de acceso a atributos irrelevantes (IAAC) de 47200 y un costo de transporte (TC) de 314500 (*best_cost_VPS*=IAAC+TC=361700). Después de actualizar el costo en **stat_performance**.*best_cost_VPS,* se dispara la **Regla 19** que

actualiza los siguientes valores en **stat_attributes**: *previousNA*= 6, *currentNA*=0, *previousChangeA*=50 %, *currentChangeA*=0, *previousChangeS*=60 %, *currentChangeS*=0. En **stat_queries** actualiza: *previousNQ*= 4, *currentNQ*=0, *previousChangeQ*=100 %, *currentChangeQ*=0, *previousChangeF*=35 %, *currentChangeF*=0. Finalmente, cambia el valor de *flag* a verdadero (*flag*=true) en **stat_performance**.

Como *flag=true*, la **Regla 20** obtiene el valor de la variable de cambio $s = 1+\frac{10}{245/4}$=1.16 y del umbral *performance_threshold=best_cost_VPS*s* =361700*1.16=419572, los almacena en **stat_performance** y llama al *Generador de Fragmentos*.

Tabla 6.1: AUM de la relación EQUIPO

Q/A	id	nombre	imagen	gráfico	audio	video	F
q_1	0	1	1	1	0	0	15
q_2	1	1	0	0	1	1	10
q_3	0	0	0	1	1	1	25
q_4	0	1	1	0	0	0	20
S	8	20	900	500	4100	39518	

Si después de la fragmentación se producen cambios en la MMDB, tenemos la AUM de la Tabla 6.2.

Tabla 6.2: AUM de la relación EQUIPO después de la fragmentación (EQUIPO2)

Q/A	id	nombre	imagen	gráfico	animación	audio	video	F
q_1	0	1	1	1	0	0	0	10
q_2	1	1	0	0	0	1	1	15
q_3	0	0	0	1	0	1	1	25
q_4	0	1	1	0	0	0	0	20
q_5	0	1	1	0	1	0	0	20
S	8	20	900	600	2000	4000	39518	

Como se inserta un nuevo atributo *animación,* se ejecuta la **Regla 1** que actualiza *currentNA*=1, lo cual a su vez dispara la **Regla 2** que calcula el porcentaje de cambio *currentChangeA*=1/6*100=16.6 %, como el porcentaje de cambio anterior *previousChangeA*

es mayor, la **Regla 3** no dispara el *Evaluador de Desempeño.*

Posteriormente se actualiza el tamaño del atributo *gráfico* de 500 bytes a 600, esto dispara la **Regla 4**, como el tamaño anterior de gráfico $prev\_size_{gráfico}$ es mayor a cero entonces se calcula el porcentaje de cambio *currentChangeS*=(600-500)/600*100=16.6 % y se actualiza en **stat_attributes**, este evento dispara las **Reglas 5, 6** y **7** cuyas acciones no se cumplen, ya que *currentChangeS* es mayor a cero y el porcentaje de cambio anterior *previousChangeS* es mayor.

Después se actualiza el tamaño del atributo *audio* de 4100 a 4000 bytes, por lo que se vuelve a disparar la **Regla 4**, la cual calcula el porcentaje de cambio *currentChangeS*=(4000-4100)/4000*100=-2.5 % y se intenta actualizar en **stat_attributes**, esto dispara las **Reglas 5, 6** y **7.** La **Regla 5** multiplica *currentChangeS**-1 debido a que *currentChangeS* es menor a cero. Como el valor de *currentChangeS* que se pretende actualizar es menor al valor que anteriormente tenía *currentChangeS,* la **Regla 6** evita que se actualice *currentChangeS*, por lo que su valor sigue siendo igual a 16.6 %, esto se hace con el objetivo de obtener el máximo porcentaje de cambio del tamaño de los atributos. La acción de la **Regla 7** no se ejecuta porque el porcentaje de cambio anterior *previousChangeS* es mayor.

A continuación se inserta una nueva consulta, por lo que se dispara la **Regla 8** que actualiza *currentNQ*=1, esto dispara la **Regla 9** que calcula *currentChangeQ*=1/4*100=25 % y lo actualiza en **stat_queries**, este evento dispara la **Regla 10** cuya acción no se realiza debido a que *previousChangeQ* es mayor.

Después se modifica la frecuencia de la primera consulta de 15 a 10, esto dispara la **Regla 11** que intenta actualizar *currentChangeF*(10-15)/15*100=-33.33 % en **stat_queries**, esto dispara las **Reglas 12**, **13** y **14**. La **Regla 12** multiplica *currentChangeF**-1 debido a que *currentChangeF* es menor a cero, las acciones de las **Regla 13** y **14** no se ejecutan porque el nuevo valor de *currentChangeF* es mayor al valor anterior (33.3>0) y *currentChangeF*<*previousChangeF.*

Por último, se modifica la frecuencia de la segunda consulta de 10 a 15, así que la **Regla 11** intenta actualizar *currentChangeF*(15-10)/10*100=50 % en **stat_queries**, este

evento dispara las **Reglas 12**, **13** y **14**. Como *currentChangeF* es mayor a cero, no se realiza la acción de la **Regla 12** y como el nuevo valor de *currentChangeF* es mayor al valor anterior tampoco se ejecuta la **Regla 13**, por lo que se actualiza *currentChangeF* y debido a que *currentChangeF* es mayor que *previousChangeF* (50 %>35 %), la **Regla 14** actualiza *flag_ cf*=true en **stat_flags**, esto provoca que la Regla 15 dispare el *Evaluador de Desempeño.*

Usando la AUM de la Tabla 6.2 y el modelo de costo propuesto en el Capítulo 4, el Evaluador de Desempeño obtiene el nuevo costo del EFV, el nuevo costo de acceso a atributos irrelevantes (IAAC) es 75700 y el nuevo costo de transporte (TC) es 747500, así que *cost*(*EFV*)=75700+747500, por lo que la **Regla 16** actualiza *cost_ VPS*=823200 en **stat_performance**. Esto provoca que la **Regla 17** dispare a MAVP debido a que *cost_ VPS*>*performance_ threshold* (823200>419572). Por lo que MAVP obtiene un nuevo EFV *(id, audio, video, gráfico), (nombre, imagen, animación)*, con un costo de 612460.

También utilizamos la MMDB **ALBUMS**(*name, birth, genre, location, picture, songs, clips*) presentada en [19]. En este caso consideramos las siguientes consultas:

q_1: Encontrar todas las canciones de hiphop

q_2: Encontrar todas las fotos de cantantes

q_3: Encontrar todos los videos y fotos de cantantes de París.

La AUM de **ALBUMS** se presenta en la Tabla 6.3. Después de la fragmentación, ocurren algunos cambios como puede verse en la Tabla 6.4. Una nueva consulta se ejecuta en la base de datos:

q_4: Encontrar el nombre y la foto de los cantantes que nacieron en 1975.

Tabla 6.3: AUM de la relación ALBUMS.

Q/A	name	birth	genre	location	picture	songs	clips	F
q_1	0	0	1	0	0	1	0	25
q_2	0	0	0	0	1	0	0	30
q_3	0	0	0	1	1	0	1	28
S	20	10	15	20	600	5000	50000	

Tabla 6.4: AUM de la relación ALBUMS después de la fragmentación (ALBUMS2).

Q/A	name	birth	genre	location	picture	songs	clips	F
q_1	0	0	1	0	0	1	0	50
q_2	0	0	0	0	1	0	0	20
q_3	0	0	0	1	1	0	1	60
q_4	1	1	0	0	1	0	0	15
S	20	10	15	20	600	5500	60000	

Los EFVs obtenidos por MAVP y DYMOND se presentan en la Tabla 6.5. Finalmente, la comparación de costo entre dichos esquemas se muestra en la Tabla 6.6. Como puede verse en la Tabla 6.6, DYMOND permite obtener EFVs con menor costo que MAVP cuando cambian los patrones de acceso y el esquema de la MMDB, ya que dispara un nuevo proceso de fragmentación si ocurren cambios mayores a los cambios promedio en la MMDB y si el costo del EFV actualmente implementado se vuelve mayor al umbral de desempeño previamente determinado por DYMOND.

Tabla 6.5: VPSs obtenidos por MAVP y DYMOND.

R	MAVP	DYMOND
E	(id, audio, video) (name, image, graphic)	(id, audio, video) (name, image, graphic)
E2	(id, audio, video) (name, image, graphic) (animation)	(id, audio, video, graphic) (name, image, animation)
A	(name, birth)(genre, songs) (location, clips) (picture)	(name, birth)(genre, songs) (location, clips)(picture)
A2	(name, birth)(genre, songs) (location, clips)(picture)	(name, birth, picture, location, clips) (genre, songs)
R=Relación, E=EQUIPO, A=ALBUMS		

6.3. Discusión

Las principales ventajas de DYMOND son:

Tabla 6.6: Comparación de costo de los EFVs obtenidos por MAVP y DYMOND.

Relación	MAVP			DYMOND		
	IAAC	TC	Costo	IAAC	TC	Costo
EQUIPO	47200	314500	361700	47200	314500	361700
EQUIPO2	69700	747500	817200	547960	64500	612450
ALBUMS	0	470400	470400	0	470400	470400
ALBUMS2	0	2166750	2166750	2103100	0	2103100

1. Mejora el desempeño total de la MMDB, ya que detecta oportunamente cuando el desempeño del sistema cae bajo un umbral para disparar el proceso de fragmentación.

2. Reduce las actividades que debe realizar el DBA, tales como el análisis de la MMDB, la determinación del EFV y la creación de los fragmentos.

3. Permite una recuperación eficiente de objetos multimedia de la MMDB en todo tiempo.

6.4. Conclusión

En este Capítulo se presentó la arquitectura de DYMOND que es un sistema activo para fragmentación vertical dinámica en bases de datos multimedia. DYMOND utiliza reglas activas para monitorizar continuamente los cambios en los patrones de acceso a la MMDB, evaluar los cambios y disparar el proceso de fragmentación permitiendo una recuperación eficiente de objetos multimedia en todo tiempo.

Capítulo 7

Implementación de DYMOND en PostgreSQL

En esta tesis doctoral se propone un método basado en reglas activas para fragmentación vertical dinámica de bases de datos multimedia, llamado DYMOND. Este método permite mejorar el tiempo de respuesta de las consultas en la base de datos.

Debido a que los algoritmos de fragmentación vertical tradicionales no consideran datos multimedia, el primer paso en el desarrollo del sistema fue crear un algoritmo de fragmentación vertical para bases de datos multimedia, llamado MAVP, el cual se presentó en el Capítulo 4.

DYMOND fragmenta una base de datos multimedia en forma automática y dinámica, para esto monitoriza continuamente la base de datos para recolectar estadísticas relacionadas a los patrones de acceso (consultas y sus respectivas frecuencias), así como el esquema de la base de datos (atributos), con base en dicha información se genera un esquema de fragmentación vertical inicial, por lo que se elimina el costo de la etapa de análisis del sistema que tenía que realizar el DBA.

Posteriormente, cuando las estadísticas sufren suficientes cambios se evalúa el costo del esquema de fragmentación vertical, para esto se desarrolló un modelo de costo para evaluar distintos esquemas de fragmentación vertical en bases de datos multimedia distribuidas,

dicho modelo se describe también en el Capítulo 4.

Si el costo del esquema es mayor que un umbral de desempeño, entonces se genera un nuevo esquema de fragmentación, para esto se utiliza MAVP, el cual primero genera una tabla de uso de atributos con base en las estadísticas y después utiliza dicha tabla para obtener un esquema de fragmentación que minimice la cantidad de atributos irrelevantes accedidos por las consultas.

El umbral de desempeño se actualiza después de cada fragmentación, este será igual al producto del costo del nuevo esquema y una variable que mide el cambio promedio en los patrones de acceso y el esquema de la base de datos multimedia.

Finalmente se materializa el nuevo esquema de fragmentación, esto es, se eliminan los fragmentos anteriores y se crean los nuevos. Los fragmentos se implementan como índices para que cada vez que un usuario realice una consulta a la base de datos multimedia, el optimizador de consultas del DBMS no tenga que reescribir la consulta, sino que utilice el índice automáticamente para responder la consulta.

DYMOND se implementa dentro del servidor de la base de datos utilizando reglas activas, por lo tanto, las reglas activas permiten monitorizar continuamente la base de datos para detectar oportunamente cambios importantes y con base en esos cambios ejecutar las acciones pertinentes.

En este Capítulo se describirán los detalles del desarrollo de DYMOND en el DBMS objeto-relacional PostgreSQL. DYMOND se implementó utilizando disparadores o triggers. Como puede verse en la Figura 7.1, el funcionamiento de DYMOND se divide en dos etapas: fragmentación vertical estática y fragmentación vertical dinámica.

7.1. Creación del EFV inicial

El primer objetivo de DYMOND es encontrar un esquema de fragmentación vertical (EFV) inicial (ver Figura 7.2), para esto utiliza la información almacenada en el log de transacciones (log SQL). En la Figura 7.3 se muestra el diagrama de secuencia que contiene los pasos necesarios para realizar la fragmentación vertical estática en una base de datos

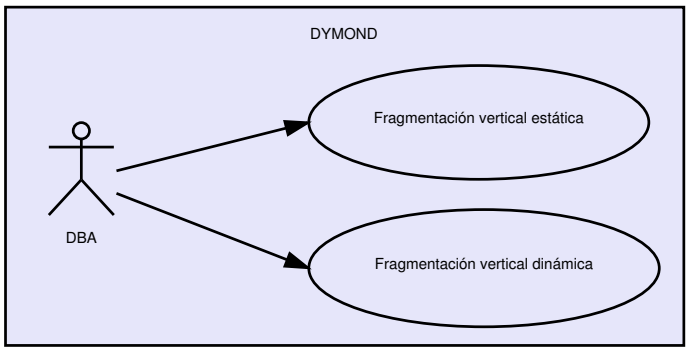


Figura 7.1: Diagrama de casos de uso de DYMOND.

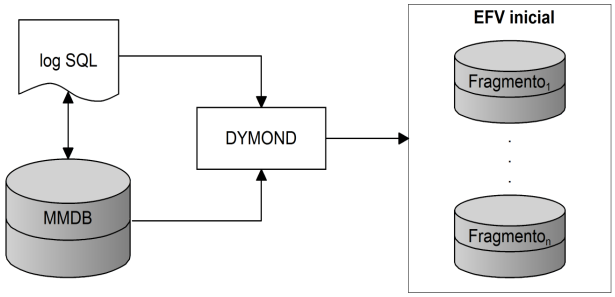


Figura 7.2: Fragmentación vertical estática desarrollada por DYMOND.

multimedia (MMDB) centralizada, y enseguida se describe cada paso.

7.1.1. Creación del esquema dymond inicial

Paso 1. El DBA ejecuta el script *dymond.sql* en la base de datos que contiene la tabla que desea fragmentar.

Paso 2. DYMOND crea el esquema *dymond* en la base de datos y dentro de este esquema genera las tablas, disparadores y funciones necesarias para iniciar el proceso de fragmentación vertical. Estos se muestran en la Figura 7.4 y se describen en la sección 7.3.

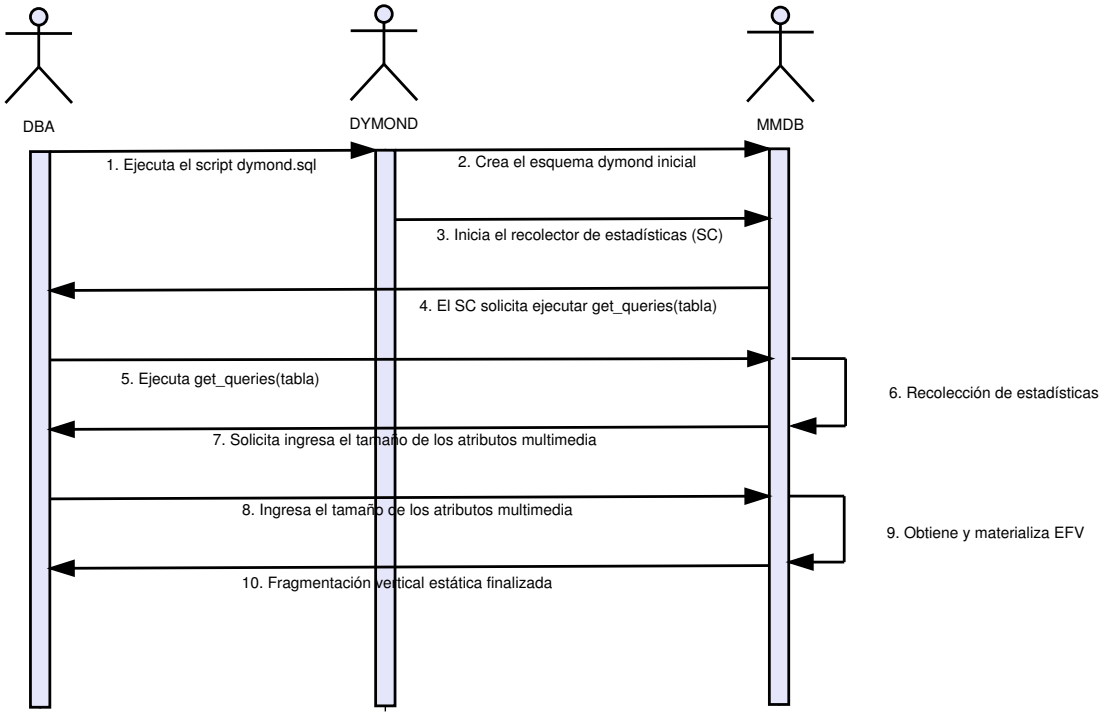


Figura 7.3: Diagrama de secuencia de la fragmentación vertical estática.

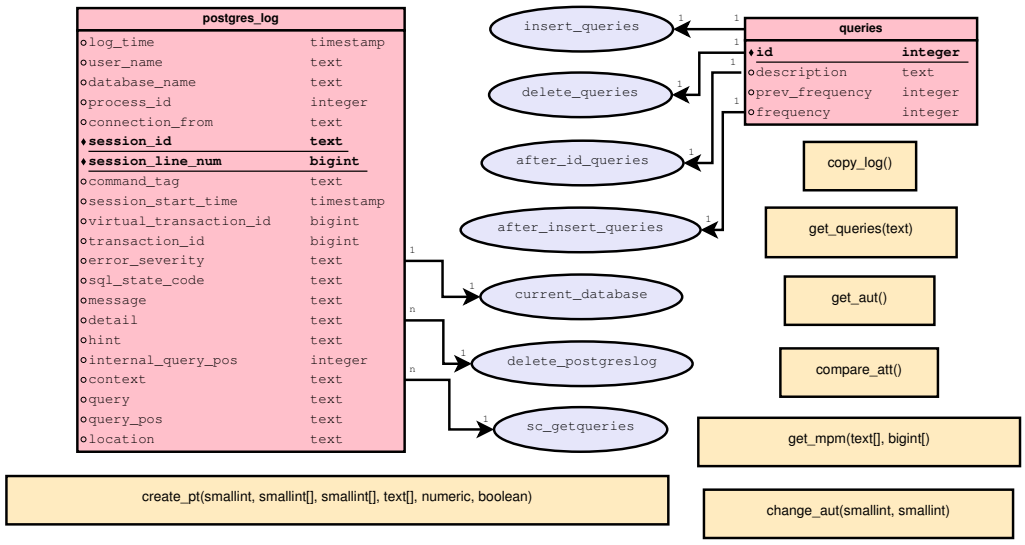


Figura 7.4: Contenido del esquema dymond en la fragmentación inicial.

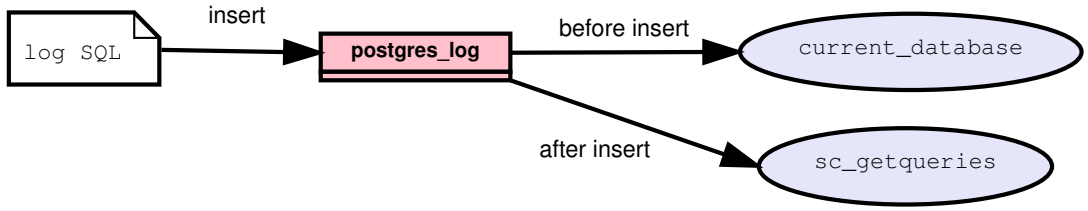


Figura 7.5: Proceso de la función copy_log().

7.1.2. Proceso de recolección de estadísticas

Paso 3. DYMOND inicia el recolector de estadísticas (SC, en inglés: Statistic Collector), para esto ejecuta la función *copy_ log()*, el proceso que realiza esta función se muestra en la Figura 7.5.

El SC de DYMOND (función *copy_ log()*) copia en la tabla **postgres_log** las consultas ejecutadas el día actual que se encuentran en el log de transacciones. Esto provoca el disparo de dos triggers.

1. Antes de insertar información sobre las consultas en la tabla **postgres_log** se dispara **current_database** que verifica que sólo se inserten las consultas realizadas en la base de datos en la que se ejecuta el script *dymond.sql* (que es la base de datos que nos interesa porque aquí se encuentra la tabla que se desea fragmentar).

2. Después de insertar toda la información sobre las consultas en la tabla **postgres_log** el disparador **sc_getqueries** verifica que exista la vista **attributes** (como es la primera vez que se ejecuta DYMOND, esta vista todavía no existe) por lo que este disparador no hace nada. El objetivo de este disparador es recolectar (almacenar en la tabla **queries**) automáticamente las consultas ejecutadas en el día actual que utilizan atributos de la tabla que se desea fragmentar (ejecutar *get_ queries(text)*).

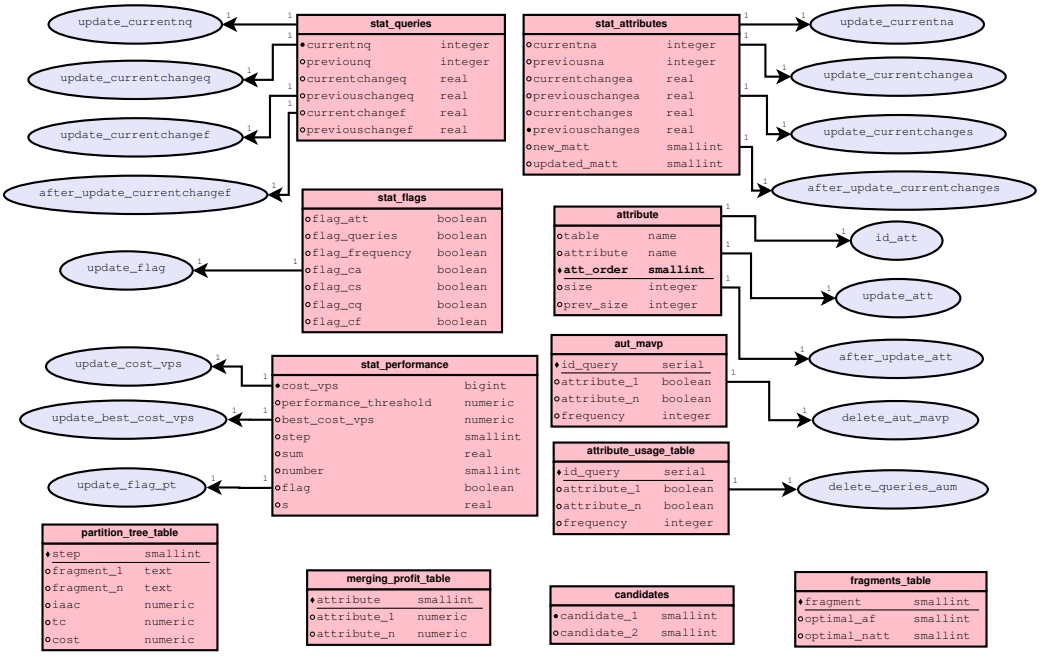


Figura 7.6: Tablas y disparadores creados por el recolector de estadísticas.

Creación de tablas y disparadores finales

Paso 4. La tabla **postgres_log** tiene todas las consultas ejecutadas en el día actual en la base de datos. Ahora el SC ya instalado en la MMDB solicita al DBA que ejecute la función *get_queries('tabla')*, donde tabla es el nombre de la tabla que desea fragmentar.

Paso 5. El DBA ejecuta la función *get_queries('tabla')*.

Paso 6. El SC continúa con la recolección de estadísticas por lo que realiza las siguientes acciones:

1. Verifica que el nombre de la tabla que ingreso el DBA sea un nombre válido; en tal caso realiza la acción número 2. En caso contrario, envía un mensaje indicando que no existe ninguna tabla en la base de datos con ese nombre y solicita al DBA que ingrese un nombre de tabla válido.

2. Genera la vista **attributes** que contiene los nombres y tamaños de todos los atributos de la tabla que se desea fragmentar.

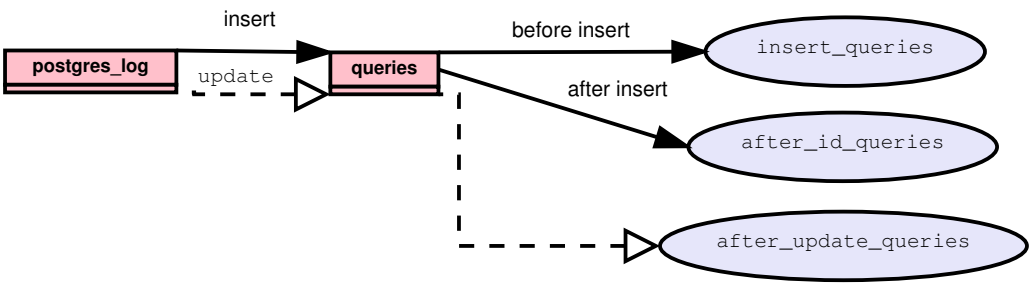


Figura 7.7: Proceso de obtención de consultas de la tabla postgres_log.

3. Crea las tablas y disparadores de la Figura 7.6 en el esquema *dymond*. En la sección 7.4 se describen cada uno de ellos.

4. EL SC copia en la tabla **queries** las consultas ejecutadas en la tabla que se desea fragmentar (almacenadas en la tabla **postgres_log**), esto dispara tres triggers (este proceso se muestra en la Figura 7.7):

- El trigger **insert_queries** que verifica si la consulta ya se encuentra en la tabla **queries** en tal caso sólo incrementa su frecuencia en 1.

- Después de insertar en queries se dispara **after_id_queries** (**Regla 8**) que verifica si no es la primera ejecución de DYMOND (**stat_attributes**.*previousnq*>0), como no se cumple esta condición entonces este disparador no hace nada.

- Cuando sólo se incrementa la frecuencia de una consulta, entonces se dispara **after_update_queries** que comprueba si no es la primera ejecución de DYMOND, como si lo es, entonces no realiza ninguna acción.

Paso 7. EL SC solicita al DBA que ingrese el tamaño de los atributos multimedia en la tabla **attribute**.

Paso 8. El DBA actualiza el tamaño de los atributos multimedia en la tabla **attribute**.

Paso 9. El SC dispara los módulos de obtención y materialización del EFV, para esto realiza las siguientes acciones:

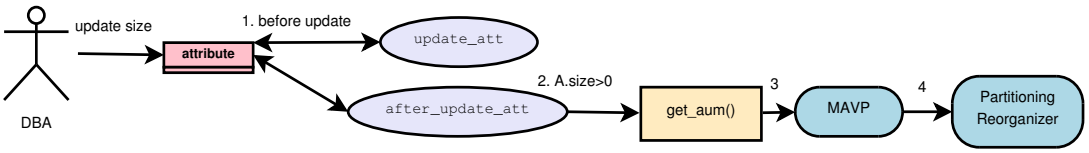


Figura 7.8: Proceso de obtención y materialización del EFV.

1. El trigger **update_att** verifica que el DBA ingrese un valor positivo y que sea la primera ejecución de DYMOND (*prev_size*=0).

2. El trigger **after_update_att** detecta el momento en el que el DBA ya ingreso todos los tamaños de los atributos multimedia (A.*size*>0) y dispara la función *get_aum()* que obtiene la matriz de uso de atributos (**attribute_usage_table** y **aut_table**). Esto sólo se realiza cuando **stat_performance**.*best_cost_vps*=0, es decir, antes de que MAVP se ejecute por primera vez.

3. Se ejecuta MAVP, el cual obtiene el EFV.

4. Se dispara el reorganizador de fragmentos cuya función es materializar el EFV.

Este proceso se muestra en la Figura 7.8.

Paso 10. Finalmente, DYMOND notifica al DBA que la MMDB se fragmentó exitosamente.

7.1.3. Determinación del EFV inicial por MAVP

En la fragmentación vertical estática MAVP es disparado por el SC (**after_update_att**). MAVP realiza lo siguiente por cada paso en la construcción de un árbol de partición:

1. Calcula el costo del EFV.

2. Obtiene la **merging_profit_table**.

3. Une el par de atributos con el mayor beneficio de fusión.

Cuando finaliza la construcción del árbol de partición:

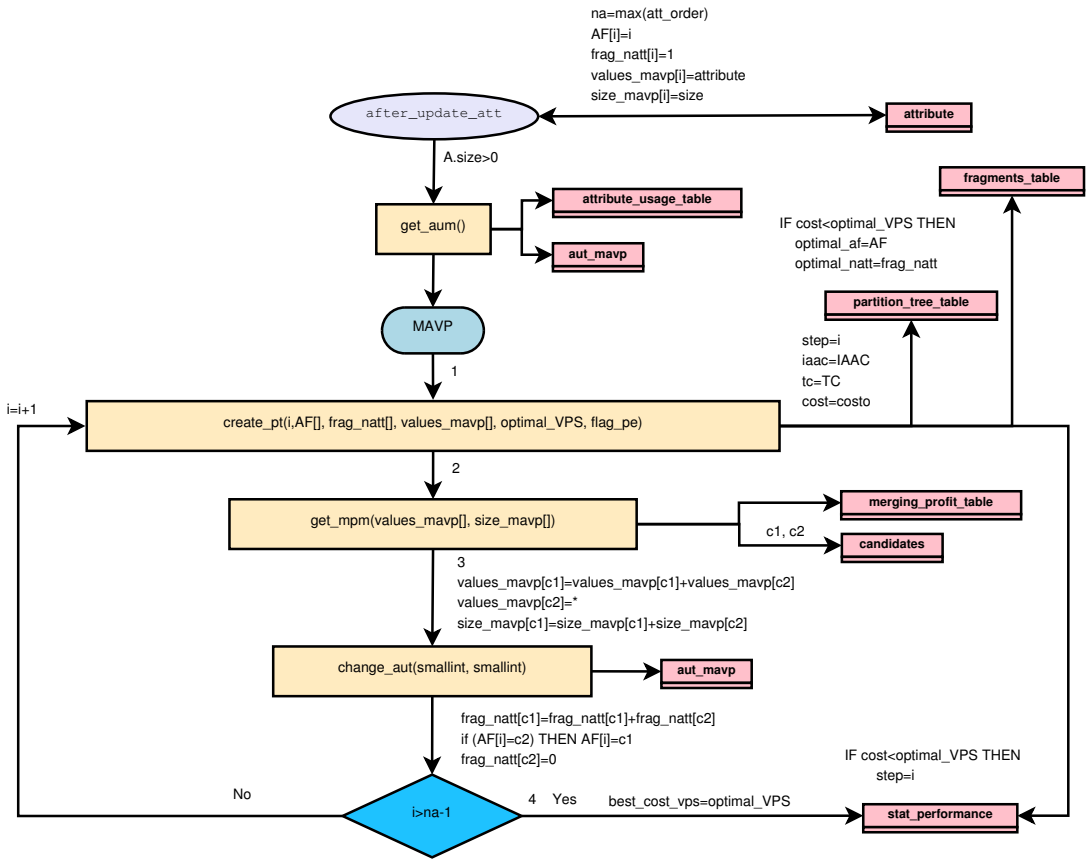


Figura 7.9: Proceso de obtención del EFV óptimo.

4. Obtiene el EFV con el costo más bajo.

Este proceso se muestra en la Figura 7. 9 y a continuación se describe cada paso.

Obtención del costo de un EFV

MAVP obtiene el costo de un EFV usando el modelo de costo presentado en el Capítulo 4, sección 5, para esto ejecuta la función *create_ pt(smallint, smallint[], smallint[], text[], numeric, boolean)* que recibe como entrada los siguientes parámetros:

- *i:* El paso que se va a calcular.
- *AF[na]:* Donde *na* es el número de atributos, este vector almacena el fragmento al que pertenece cada atributo.

- *frag_ natt[na]*: Este vector almacena el número de atributos que contiene cada fragmento.
- *values_ mavp[na]*: Este vector almacena el nombre de los atributos almacenados en cada fragmento.
- *optimal_ VPS*: Almacena el costo más bajo de todos los pasos.
- *flag_ pe*: Es la bandera que indica si la función de obtención del costo de un EFV es disparada por el evaluador de desempeño (*flag_ pe=true*) o por MAVP (*flag_ pe=false*).

El costo de un EFV se compone de costo de transporte (TC) y costo de acceso a atributos irrelevantes (IAAC).

Obtención del TC Primero calcula el costo de transporte del EFV, para esto utiliza el vector *AF*, de acuerdo a este vector, por cada consulta almacena en el vector *FS* el tamaño de los atributos ubicados en un fragmento y utilizados por la consulta, esto lo hace con el fin de determinar donde localizar cada consulta. Por ejemplo, si todos los atributos de la tabla **EQUIPO**(*id, nombre, imagen, gráfico, audio, video*) están en un fragmento diferente, el vector *AF* resultante se muestra en la Figura 7.10. El vector FS de la consulta q_1=*Encuentra todas las imágenes y gráficos de podadoras* se presenta en la misma figura. MAVP ubica las consultas en el fragmento que contiene el atributo de mayor tamaño utilizado por dichas consultas, en este caso es el fragmento 3, ya que este fragmento contiene el atributo *imagen* con 900 bytes.

Después utiliza el vector *FS* para obtener el costo de transporte, lo que hace es recorrer este vector acumulando en TC la frecuencia al cuadrado de la consulta y multiplicando por el tamaño de los atributos almacenados en *FS*, con excepción del fragmento donde se ubica la consulta. En el ejemplo anterior el costo de transporte de la consulta q_1 es igual a la frecuencia de dicha consulta al cuadrado (15^2=225), multiplicada por el tamaño de los atributos utilizados por q_1 y ubicados en un fragmento diferente al fragmento 3 (que es donde se ubicó a q_1 anteriormente), por lo tanto, TC(q_1)=225*(20+500)=117000.

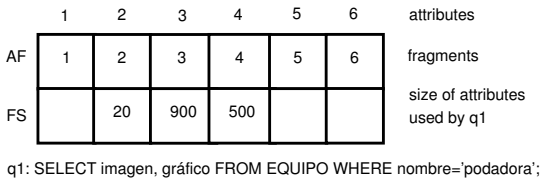


Figura 7.10: Vectores AF y FS para ubicación de consultas.

Obtención del IAAC Posteriormente calcula el costo de acceso a atributos irrelevantes, para esto utiliza los vectores *frag_ natt* y *AF*. Primero recorre *frag_ natt*; cuando un fragmento tiene más de un atributo, entonces recorre el vector *AF,* para determinar qué atributos se encuentran en el fragmento, después evalúa todas las consultas para saber si usan algún atributo del fragmento (atributos relevantes) y no usan algún atributo del fragmento (atributos irrelevantes), además acumula el tamaño de cada atributo localizado en un fragmento y no utilizado por la consulta. Si la consulta utiliza por lo menos un atributo y no utiliza por lo menos un atributo, entonces el IAAC de ese fragmento se acumulará y se le agregará la frecuencia de la consulta multiplicada por el tamaño de los atributos que no usa. Por ejemplo, si los atributos *id*, *audio* y *video* de la tabla **EQUIPO** se fusionan en un fragmento, los vectores *frag_ natt* y *AF* resultantes se muestran en la Figura 7.11. Como el primer fragmento tiene más de un atributo, entonces se recorre *AF* buscando aquellos atributos que se encuentren en el primer fragmento (*id, audio y video*), cuando se evalúa la consulta q_3= *Encuentra todos los gráficos, audio y video*, como q_3 no usa *id* y si usa *audio* y *video,* se multiplica su frecuencia por el tamaño de los atributos que no utiliza, por lo tanto, IAAC(q_3)=25*8=200.

Después de obtener el IAAC de un fragmento, si la función es disparada por MAVP almacena en **partition_tree_table** los atributos ubicados en ese fragmento, para esto utiliza el vector *values_ mavp*.

Finalmente, actualiza *iaac*, *tc*, y *cost* en la tabla **partition_tree_table**, y compara

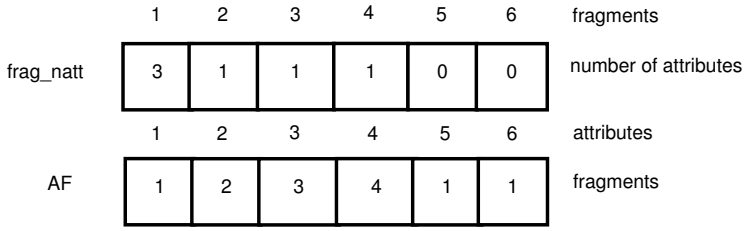


Figura 7.11: Vectores frag_natt y AF para obtención del IAAC.

el costo anterior (*optimal_ VPS*) con el nuevo costo calculado (*cost*), si *cost* es menor, entonces a *optimal_ VPS* se le asigna el valor de *cost* y se regresa *optimal_ VPS*.

Obtención de la tabla de beneficio de fusión

MAVP utiliza la tabla de beneficio de fusión para determinar que atributos unir en cada paso durante la construcción del árbol de partición. Por lo que después de calcular el costo de un EFV dispara la función *get_ mpm(text[], bigint[])*. Esta función recibe como parámetros:

- *values_ mavp[na]*: Almacena los nombres de los atributos almacenados en cada fragmento.
- *size_ mavp[na]*: Almacena el tamaño de los atributos almacenados en cada fragmento.

Primero evalúa cada par de atributos diferente, por ejemplo, (*id,nombre)* y (*nombre, id*) son el mismo par, por lo que sólo evalúa el primer par. Si una consulta usa los dos atributos juntos, entonces Q1 (el número de consultas que reduce el acceso a atributos remotos) se incrementa en uno y DRA (la cantidad reducida de atributos remotos accedidos) se acumula y se le agrega la frecuencia de la consulta multiplicada por el tamaño del atributo más pequeño; si la consulta usa uno de los atributos, entonces Q2 (el número de consultas que incrementan el acceso a atributos irrelevantes) se incrementa en 1 y IIA (la cantidad incrementada de atributos irrelevantes accedidos) se acumula y se le agrega la frecuencia de la consulta multiplicada por el tamaño del atributo que no utiliza.

Al finalizar de evaluar todas las consultas obtiene el merging profit multiplicando Q1 por DRA y a esto se le resta el valor de Q2 por IIA.

Después se almacena el merging profit en **merging_profit_table**, y se compara el merging profit obtenido con el mayor merging profit *(great_mp)*, si el nuevo es mayor, entonces a *great_mp* se le asigna el valor del merging profit obtenido y se actualiza en la tabla **candidates**, *c1* igual al primer atributo evaluado y *c2* igual al segundo atributo evaluado.

Unión del par con mayor beneficio de fusión

Una vez que *get_mpm(text[], bigint[])* determina el par de fragmentos a fusionar, MAVP cambia *values_mavp[c1]*, le agrega el nombre de los atributos almacenados en *values_mavp[c2]* y a *values_mavp[c2]* le asigna un asterisco (indicando que ese fragmento ya no contiene ningún atributo).

Posteriormente MAVP cambia *size_mavp[c1]*, le agrega el tamaño del atributo almacenado en *size_mavp[c2]* (el tamaño del fragmento fusionado ahora es igual al tamaño que los dos fragmentos tenían antes de fusionarse).

Después MAVP cambia la tabla **aut_mavp**, para esto ejecuta la función *change_aut(smallint, smallint)*. Esta función recibe como parámetros:

- *attribute_1* :Es el valor del primer atributo fusionado (*c1*).
- *attribute_2* :Es el valor del segundo atributo fusionado (*c2*).

Si una consulta usa *attribute_1* o *attribute_2*, entonces ahora la consulta usa *attribute_1* y ya no usa *attribute_2*.

El número de atributos que contiene el fragmento *c1* (*frag_natt[c1]*) ahora es igual al número de atributos que contenía el fragmento *c1* mas el número de atributos que contiene el fragmento *c2* (*frag_natt[c1]=frag_natt[c1]+frag_natt[c2]*). El número de atributos almacenado en el fragmento *c2* ahora es igual a cero (*frag_natt[c2]=0*).

MAVP cambia el vector *AF,* de tal forma que a los atributos almacenados en el fragmento *c2* (*AF[i]=c2*) ahora les asigna el fragmento *c1* (*AF[i]=c1*).

Obtención del EFV óptimo

Al finalizar la construcción del árbol de partición (*i>na-1*), MAVP actualiza *best_ cost_ vps* en la tabla **stat_performance** (*best_ cost_ vps=optimal_ VPS*).

Antes de actualizar *best_ cost_ vps* en la tabla **stat_performance**, se dispara el trigger **update_best_cost_vps**, el cual realiza lo siguiente:

1. Actualiza los valores *previousna* y *previousnq* en **stat_attributes** y **stat_queries** respectivamente, a *previousna* le asigna el número de atributos, y a *previousnq* le asigna el número de consultas.

2. Actualiza *prev_freq* de todas las consultas en la tabla **queries**, asignándole el valor de *frequency* y a *frequency* le asigna un cero.

3. Actualiza *prev_ size* de todos los atributos en la tabla **attribute**, asignándole el valor de *size*.

4. Verifica si es la primera ejecución de MAVP, como si es, entonces le asigna a *performance_ threshold* el valor de *best_ cost_ vps*.

Este proceso se muestra en la Figura 7.12.

7.1.4. Materialización del EFV inicial

Finalmente, después de ejecutar MAVP, el disparador **after_update_att** ejecuta el *Generador de Fragmentos* (función *partitioning_ generator()*). Los fragmentos se implementan en DYMOND como índices.

Los pasos que realiza el *Generador de Fragmentos* son:

1. Recupera el número de atributos (*n_ att*) y el nombre de la tabla que se va a fragmentar (*t_ name*) de la vista *attributes*.

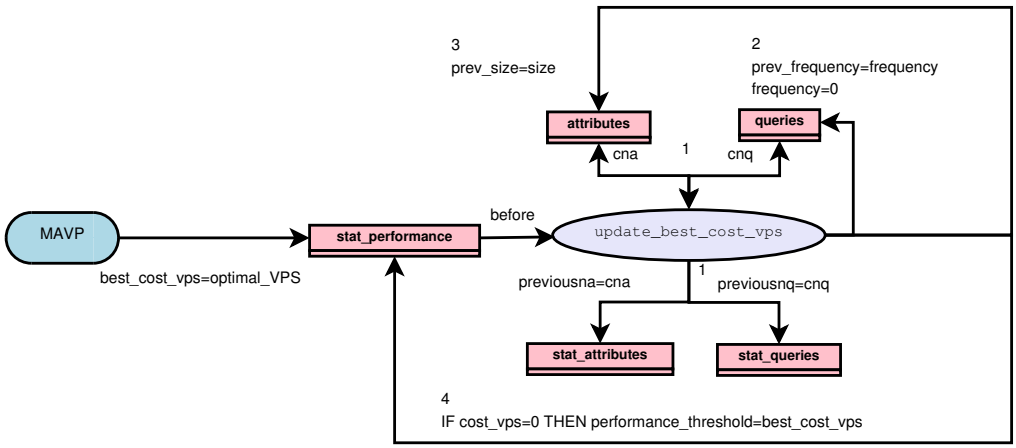


Figura 7.12: Proceso de cambio de estadísticas al obtener un EFV.

2. Encuentra la clave primaria (*cpkey*) de la tabla que se desea fragmentar, para lograrlo obtiene el nombre de la columna (*column_name*) del catálogo ANSI (*information_schema*), del objeto del catálogo *key_column_usage,* donde *table_name*='*t_name*' y *constraint_name*='*t_name_pkey*'.

3. Crea los fragmentos (índices), para esto selecciona el nombre de cada atributo de la vista *attributes*, si el atributo no es la clave primaria de la tabla, entonces determina en qué fragmento se encuentra el atributo (su ubicación se encuentra almacenada en **fragments_table**.*optimal_af*) y se almacena dentro del arreglo *query*[*n_att*] los atributos que se ubican en cada fragmento. Posteriormente, se eliminan los índices anteriores. Finalmente, si el fragmento contiene atributos (*query[i]* no es nulo) crea el índice, para esto utiliza el nombre de la tabla (*t_name*); el nombre de la clave primaria, la cual se agrega a todos los fragmentos; y los atributos almacenados en el arreglo *query*, *query[i]*.

En la Figura 7.13 se presenta este proceso.

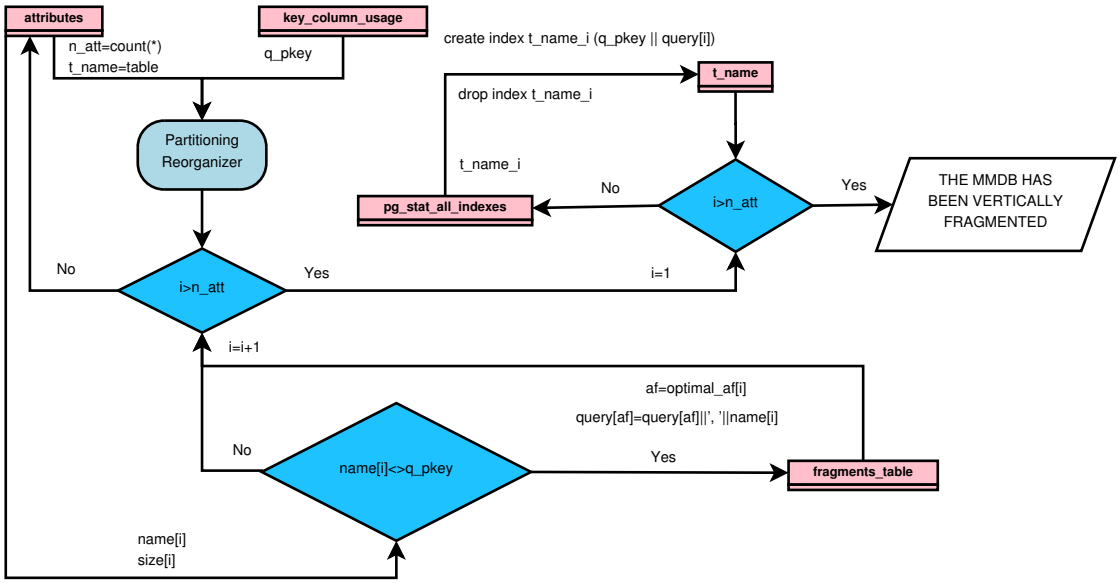


Figura 7.13: Proceso de generación de fragmentos.

7.2. Proceso de la fragmentación vertical dinámica

DYMOND utiliza un disparador temporal para realizar la fragmentación vertical dinámica, en PostgreSQL lo implementa utilizando pgAgent. PgAgent es un programador de tareas para PostgreSQL que puede administrarse usando pgAdmin. En la Figura 7.14 se presenta el proceso de fragmentación vertical dinámica en DYMOND y la Figura 7.15 muestra el diagrama de secuencia de dicho proceso.

Paso 1. Se crean los disparadores que se van a ejecutar diariamente a la 23:58, para esto el DBA ejecuta el script *dymond_jobs.sql* en la base de datos postgres.

Paso 2. El script *dymond_jobs.sql* crea el esquema dymond_jobs y dentro de este esquema crea la función *create_jobs('text')*.

Paso 3. La MMDB solicita al DBA ejecutar la función *create_jobs(database)*, donde *database* es el nombre de la base de datos que contiene la tabla que se desea fragmentar.

Paso 4. El DBA ejecuta la función *create_jobs(database)*.

Paso 5. DYMOND ya instalado en la MMDB crea el disparador **statistic_collector** que iniciará el recolector de estadísticas todos los días a las 23:58.

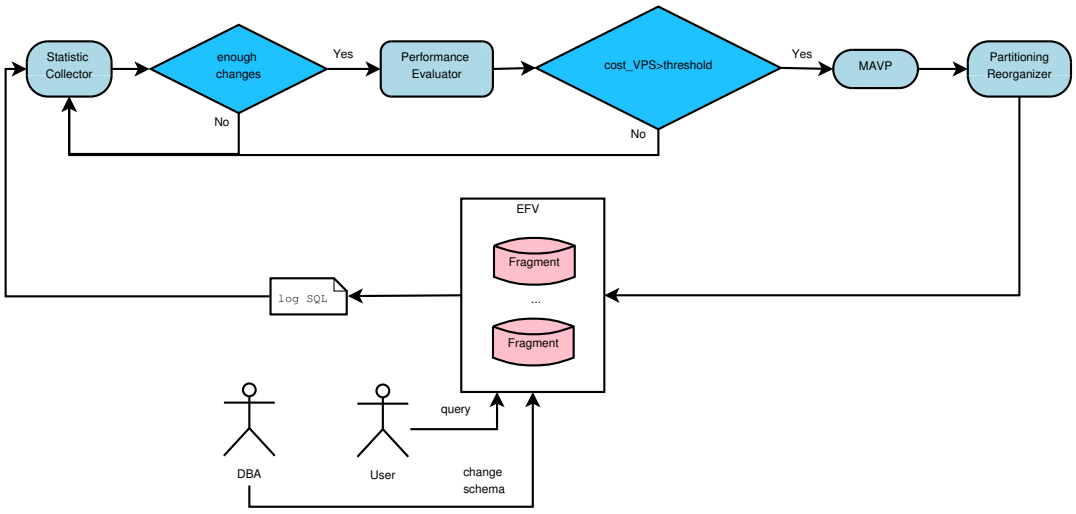


Figura 7.14: Proceso de fragmentación vertical dinámica en DYMOND.

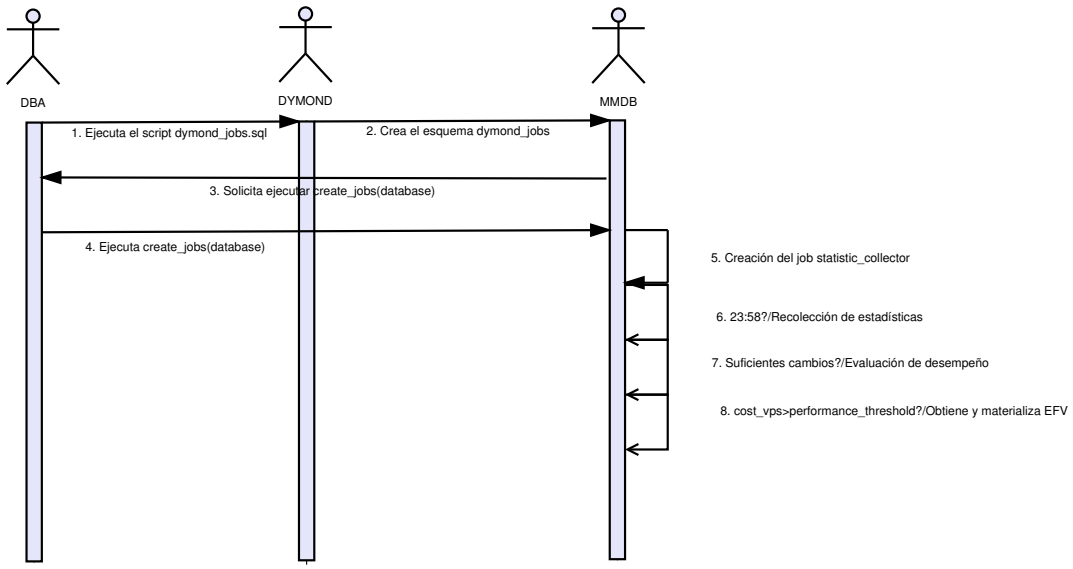


Figura 7.15: Diagrama de secuencia de la fragmentación vertical dinámica.

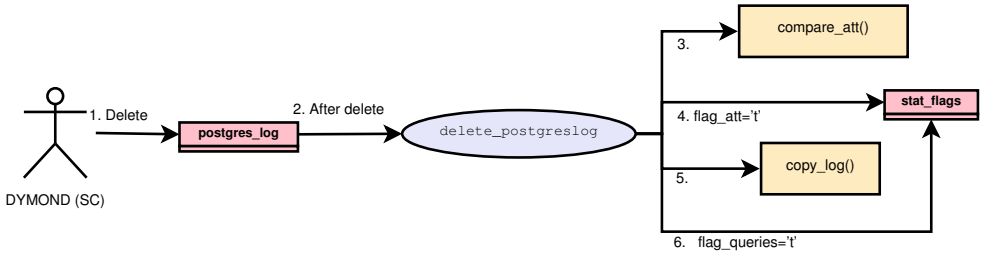


Figura 7.16: Proceso de recolección de estadísticas en la fragmentación vertical dinámica.

7.2.1. Recolección de estadísticas

Paso 6. El recolector de estadísticas (SC) realiza las siguientes acciones:

1. Elimina las consultas del día anterior de la tabla **postgres_log.**
2. La eliminación de los datos de **postgres_log** dispara el trigger **delete_postgreslog**.
3. El trigger **delete_postgreslog** ejecuta la función *compare_att()* para determinar si hay cambios en los atributos suficientes para disparar el evaluador de desempeño.
4. Cambia *flag_att* a verdadera en **stat_flags** para indicar que ya se evaluaron todos los atributos.
5. El trigger **delete_postgreslog** ejecuta la función *copy_log()* para determinar si hay cambios en las consultas suficientes para disparar el evaluador de desempeño.
6. Cambia *flag_queries* a verdadera en **stat_flags** para indicar que ya se evaluaron todas las consultas.

Este proceso se ilustra en la Figura 7.16. El proceso de las funciones *compare_att()* y *copy_log()* se describen en las siguientes subsecciones.

7.2.2. Determinación de cambios en los atributos

La función *compare_att()* compara la vista **attributes** con la tabla **attribute**, con el objetivo de determinar si se agregaron, eliminaron o modificaron los atributos de la tabla

fragmentada.

Si el DBA realiza cambios en los atributos, pueden ocurrir los siguientes casos:

1. INSERCIÓN DE ATRIBUTOS. En caso de que el DBA agrege nuevos atributos a la tabla fragmentada, estos se agregan en forma automática a la vista **attributes**, la función *compare_ att()* detecta los nuevos atributos en la vista **attributes** y los inserta en la tabla **attribute**, esto genera el disparo de **id_att** (**Regla 1**) que agrega una columna con el nombre del nuevo atributo en las tablas **attribute_usage_table**, **aut_mavp** y **merging_profit_table.** Además, si el nuevo atributo es multimedia, entonces se pide al DBA que actualice su tamaño en la tabla **attribute** e incrementa en 1 *new_ matt* y *updated_ matt* en **stat_attributes.** Si el nuevo atributo no es multimedia, entonces incrementa *currentna* en 1 en la tabla **stat_attributes** por cada atributo insertado. Este proceso se muestra en la Figura 7.17.

2. ELIMINACIÓN DE ATRIBUTOS. En caso de que el DBA elimine atributos de la tabla fragmentada, estos se eliminan en forma automática de la vista **attributes,** la función *compare_ att()* detecta los atributos eliminados y los borra de la tabla **attribute**, esto genera el disparo de **id_att** (**Regla 1**) que incrementa *currentna* en 1 por cada atributo eliminado y quita la columna con el nombre del atributo eliminado de las tablas **attribute_usage_table**, **aut_mavp** y **merging_profit_table**. Este proceso se muestra en la Figura 7.17.

3. ACTUALIZACIÓN DE ATRIBUTOS. En caso de que el DBA cambie el nombre o el tamaño de los atributos, se actualiza el nombre o el tamaño en la tabla **attribute**, esto dispara **update_att** que realiza acciones de acuerdo a los siguientes casos:

   - Si se actualizó el nombre de un atributo, entonces cambia el nombre en las tablas **attribute_usage_table**, **aut_mavp** y **merging_profit_table**.
   - Si el nuevo tamaño del atributo es positivo y no es la primera ejecución de DYMOND (**stat_performance**.*best_ cost_ vps*>0), si se actualiza un atributo

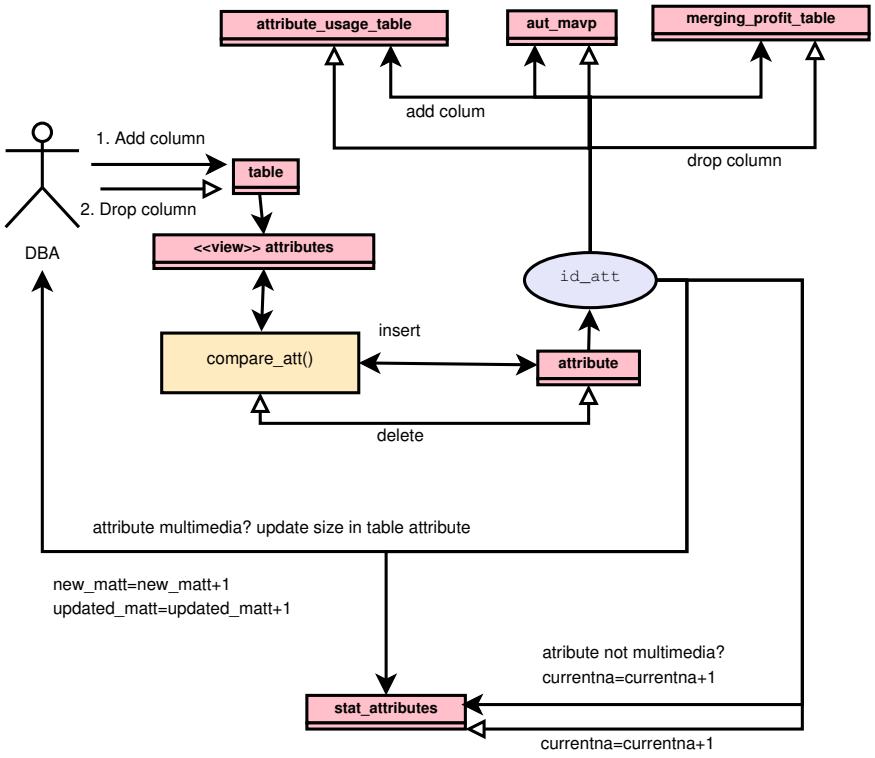


Figura 7.17: Proceso de inserción y eliminación de atributos.

multimedia (old.*size*=-5), entonces reduce en 1 el número de atributos multimedia actualizados (**stat_attributes**.*updated_matt*) y si dicho número es igual a cero, incrementa *currentna* en **stat_attribute**s con el número de atributos multimedia insertados (*new_matt*). Si el atributo modificado no es multimedia (old.*size*<>-5), entonces actualiza el valor de *currentchanges* en la tabla **stat_attributes** (**Regla 4**).

- Si el nuevo tamaño del atributo es positivo y el tamaño anterior (*prev_size*) es cero, entonces sólo se regresa el nuevo valor, ya que es un atributo multimedia actualizado por el DBA.
- Si se actualiza el tamaño de un atributo multimedia (new.*size*=-5), entonces le deja el tamaño anterior, ya que el DBA ya ingreso su valor.
- Si se actualiza a un tamaño negativo (diferente a -5), entonces despliega un mensaje de que el tamaño del atributo no puede ser negativo.

Después de actualizar en la tabla attribute se dispara **after_update_att** que verifica si se actualizó el tamaño de un atributo multimedia (-5 a positivo). En tal caso, verifica si es la primera ejecución; como esto no se cumple, entonces este disparador no hace nada. Este proceso se muestra en la Figura 7.18.

Finalmente, al realizar cambios en la tabla **stat_attributes** se tienen los siguientes casos:

1. ELIMINACIÓN/INSERCIÓN DE ATRIBUTOS. Cuando se actualiza en la tabla **stat_attributes,** el disparador **update_currentna** (**Regla 2**) verifica que el valor actualizado sea *currentna.* En tal caso, verifica que *currentna* y *previousna* sean mayores a cero (esto se hace con el objetivo de que no se ejecute la acción cuando se actualice *currentna* a 0 después de una nueva fragmentación); si es así, obtiene el valor de *currentchangea*. Después de modificar el valor de *currentchangea* en la tabla **stat_attribute** se dispara **update_currentchangea** (**Regla 3**) que verifica si *currentchangea* es mayor que *previouschangea.* En tal caso, cambia *flag_ca* a

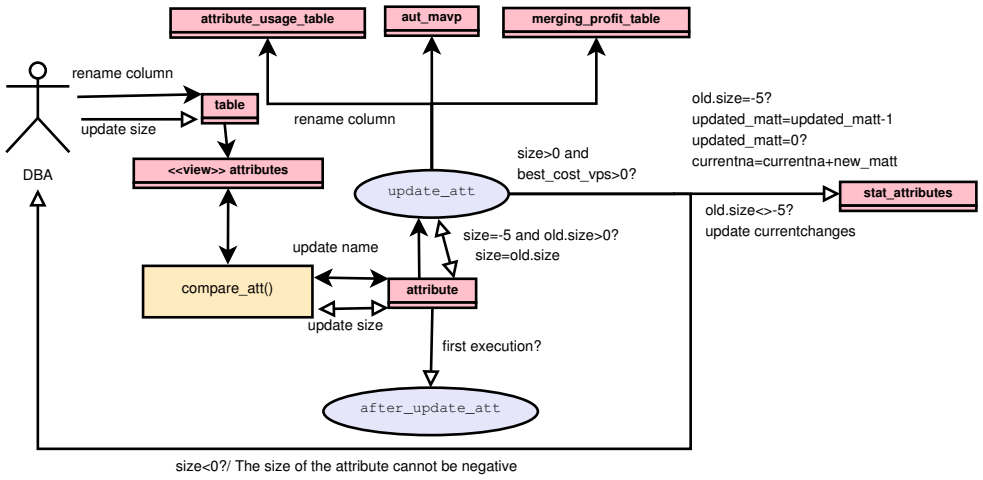


Figura 7.18: Proceso de actualización de atributos.

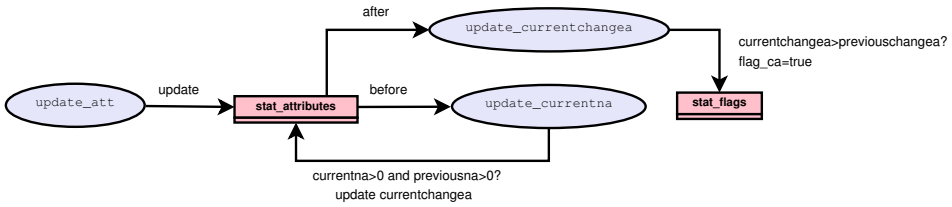


Figura 7.19: Proceso de cambios en el número de atributos.

verdadera en la tabla **stat_flags**. Este proceso se muestra en la Figura 7.19.

2. ACTUALIZACIÓN DEL TAMAÑO DE LOS ATRIBUTOS. Cuando se actualiza en la tabla **stat_attribute,** el disparador *update_currentchanges* (**Regla 5**) verifica que el valor actualizado sea *currentchanges.* En tal caso, si el valor actualizado es negativo, entonces lo multiplica por -1 para hacerlo positivo. Si el valor actualizado es menor al valor anterior, entonces se deja el valor anterior (**Regla 6**). Después de modificar el valor de *currentchanges* en la tabla **stat_attribute,** se dispara *after_update_currentchanges* (**Regla 7**) que cambia *flag_cs* a verdadera en la tabla **stat_flags** si *currentchanges* es mayor que *previouschanges*. Este proceso se muestra en la Figura 7.20.

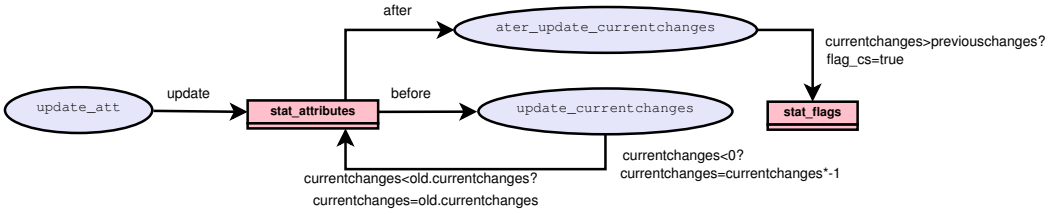


Figura 7.20: Proceso de cambios en el tamaño de los atributos.

7.2.3. Determinación de cambios en las consultas

Para evaluar los cambios en las consultas, el SC realiza las siguientes acciones:

1. La función *copy_ log()* copia las consultas ejecutadas el día actual que se encuentran en el log de transacciones en la tabla **postgres_log**.

2. Antes de insertar información sobre las consultas en la tabla **postgres_log,** se dispara **current_database** que verifica que sólo se inserten las consultas realizadas en la base de datos actual.

3. Después de insertar toda la información sobre las consultas en la tabla **postgres_log,** el dispardor **sc_getqueries** verifica que exista la vista **attributes**; como no es la primera vez que se ejecuta DYMOND, esta vista ya existe, por lo que este disparador selecciona el nombre de la tabla que se esta fragmentando dinámicamente de la vista **attributes** y recolecta (almacena en la tabla **queries**) automáticamente las consultas ejecutadas en el día actual sobre la tabla que se desea fragmentar (ejecuta *get_ queries(tabla)*).

   Este proceso se muestra en la Figura 7.21.

La función *get_ queries(tabla)* realiza las siguientes acciones:

1. Copia en la tabla **queries** las consultas ejecutadas en la tabla que se desea fragmentar (almacenadas en la tabla **postgres_log**).

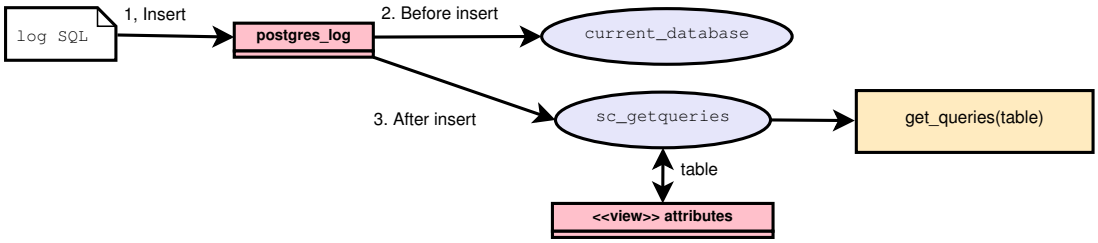


Figura 7.21: Proceso de la función copy_log en la fragmentación vertical dinámica.

2. Antes de insertar una consulta en la tabla **queries**, se dispara el trigger **insert_queries** que verifica si la consulta ya se encuentra en la tabla **queries;** en tal caso sólo incrementa su frecuencia en 1.

3. Después de insertar en **queries,** se dispara **after_id_queries** (**Regla 8**) que verifica si no es la primera ejecución de DYMOND (**stat_queries**.*previousnq*>0), como si se cumple esta condición, entonces este disparador incrementa el valor de *currentnq* en 1 por cada consulta insertada.

4. Cuando se actualiza en la tabla **stat_queries,** el disparador **update_currentnq** (**Regla 9**) verifica que el valor actualizado sea *currentnq;* en tal caso verifica que *currentnq* y *previousnq* sean mayores a cero (esto se hace con el objetivo de que no se ejecute la acción cuando se actualice *currentnq* a 0 después de una nueva fragmentación); si es así, obtiene el valor de *currentchangeq*.

5. Después de actualizar en la tabla **stat_queries,** el disparador **update_currentchangeq** (**Regla 10**) verifica que *currentchangeq* sea mayor que *previouschangeq;* en tal caso cambia *flag_cq* a verdadera en la tabla **stat_flags**.

Este proceso se muestra en la Figura 7.22.

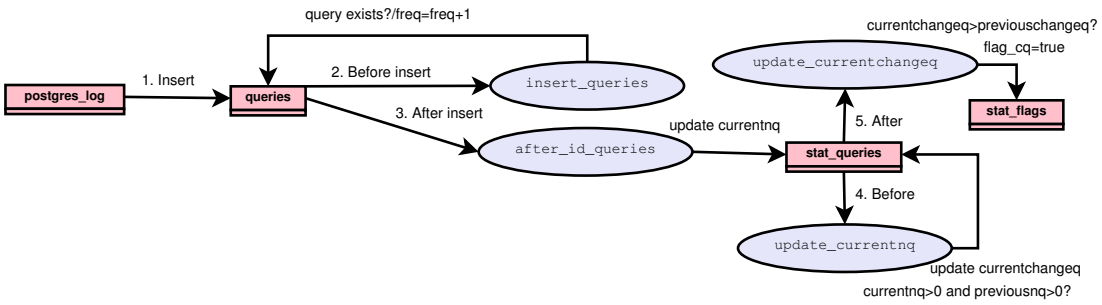


Figura 7.22: Proceso de la función get_queries(tabla) en la fragmentación vertical dinámica.

7.2.4. Determinación de cambios en la frecuencia de las consultas

El SC realiza las siguientes acciones para evaluar los cambios en la frecuencia de las consultas:

1. Después de actualizar *flag_ queries=true* en **stat_flags,** el trigger **delete_postgres-log** (**Regla 11**) calcula el porcentaje de cambio en la frecuencia de las consultas (*currentchangef*) y lo actualiza en **stat_queries**, finalmente cambia *flag_frequency* a verdadera en **stat_flags** para indicar que ya se calculó el porcentaje de cambio en la frecuencia de todas las consultas.

2. Cuando se actualiza el valor de *currentchangef* en la tabla **stat_queries**, el trigger **update_currentchangef** (**Reglas 12 y 13**) multiplica *currentchangef* por -1 si *currentchangef* es negativo y si el valor actualizado de *currentchangef* es menor que el valor anterior de *currrentchangef* (new.*currentchangef*<old.*currentchangef*) entonces deja el valor anterior.

3. Después de actualizar el valor de *currentchangef* en la tabla **stat_queries**, si el valor actualizado es mayor que el valor de cambio en la frecuencia de las consultas de la última fragmentación entonces, el trigger **after_update_currentchangef** (**Regla 14**) cambia *flag_ cf* a verdadera en la tabla **stat_flags**.

Este proceso se muestra en la Figura 7.23.

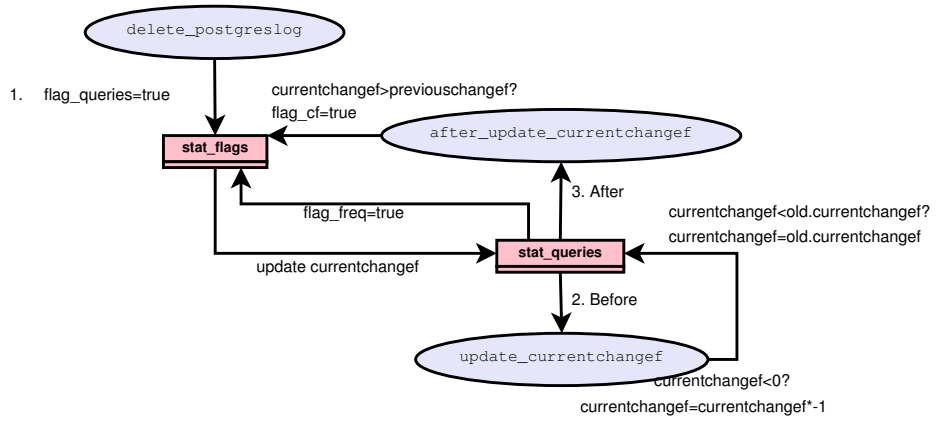


Figura 7.23: Proceso de evaluación de cambios en la frecuencia de las consultas.

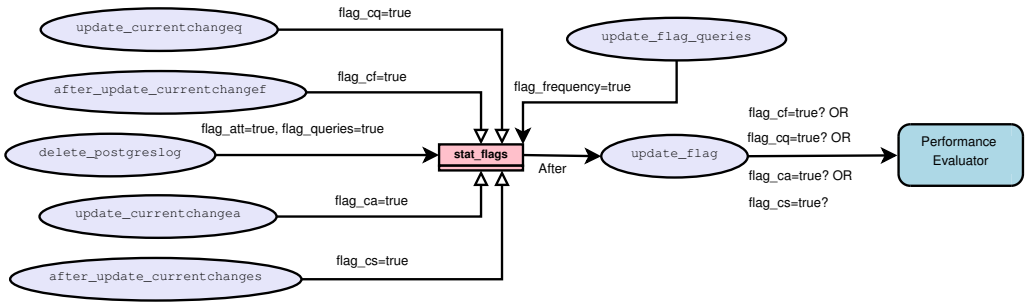


Figura 7.24: Proceso de disparo del evaluador de desempeño.

Finalmente, después de actualizar en la tabla **stat_flags,** si ya se evaluaron todos los atributos (*flag_att=true*), todas las consultas (*flag_queries=true*) y la frecuencia de todas las consultas (*flag_frequency=true*), entonces el trigger **update_flag** (**Regla 15**) determina si los cambios disparan el *Evaluador de Desempeño*. Esto se muestra en la Figura 7.24.

7.2.5. Evaluación de desempeño del EFV

Los pasos que realiza el *Evaluador de Desempeño* cuando es disparado por el *Recolector de Estadísticas* (**update_flag**) son los siguientes:

1. Recupera información del EFV a evaluar.

2. Obtiene la nueva **attribute_usage_table**.

3. Calcula el nuevo costo del EFV.

4. Actualiza el costo en **stat_performanc**e.

Recuperación de información del EFV a evaluar

El paso (*step*) almacenado en **stat_performance** es el EFV actualmente implementado, por lo que el *Evaluador de Desempeño* obtiene este paso (*best_vps*).

Para calcular el costo del EFV, el *Evaluador de Desempeño* debe recuperar de **fragments_table** la información sobre los fragmentos en los que se encuentra cada atributo (*optimal_af*) y el número de atributos que contiene cada fragmento (*optimal_natt*) y almacenar dicha información en los vectores AF y *frag_natt* respectivamente.

También debe recuperar de **partition_tree_table** el nombre de los atributos ubicados en cada fragmento (en el paso obtenido de **stat_performance**) y guardar esta información en el vector *values_att.*

Obtención de la tabla de uso de atributos

El *Evaluador de Desempeño* debe obtener la tabla de uso de atributos usando la información de la tabla **queries** para calcular el nuevo costo del EFV, para esto ejecuta la función *get_aum()*, la cual realiza lo siguiente:

Verifica si **attribute_usage_table** tiene datos. Como si los tiene, ya que no es la primera fragmentación, entonces elimina todos los datos de **attribute_usage_table** y de **aut_mavp**, y analiza las consultas almacenadas en la tabla **queries** para determinar cuales atributos usan y cuales no. Finalmente, guardan esta información en **attribute_usage_table** y **aut_mavp**.

Cálculo del nuevo costo del EFV

Después de obtener la tabla de uso de atributos, el *Evaluador de Desempeño* ejecuta la función *create_pt(best_vps, AF, frag_natt, values_att, p_threshold, flag_pe)*, la cual realiza lo siguiente:

Recibe como entrada los parámetros:

- *best_vps:* Es el paso obtenido de **stat_performance**, el EFV óptimo determinado por MAVP.
- *AF:* Contiene los fragmentos en los que se ubica cada atributo actualmente.
- *frag_natt:* Contiene el número de atributos que tiene cada fragmento actualmente.
- *values_att:* Contiene el nombre de los atributos ubicados en cada fragmento actualmente.
- *p_threshold:* Este es el umbral de desempeño con el que va a comparar el costo obtenido, este valor se va con cero con el objetivo de que el costo siempre sea menor que este umbral.
- *flag_pe:* Es la bandera que indica si la función de obtención de costo del EFV es disparada por el evaluador de desempeño (*flag_pe=true*) o por MAVP (*flag_pe=false*).

Obtiene el costo de transporte (TC) y de acceso a atributos irrelevantes (IAAC), suma este costo y lo compara con *p_threshold* que es igual a cero, como el costo es mayor y *flag_pe=true*, entonces almacena el costo en *new_cost* y lo regresa al evaluador de desempeño.

Actualización del costo en las estadísticas de desempeño

Finalmente, el *Evaluador de Desempeño* actualiza este costo (*cost_vps=new_cost*) en **stat_performance**.

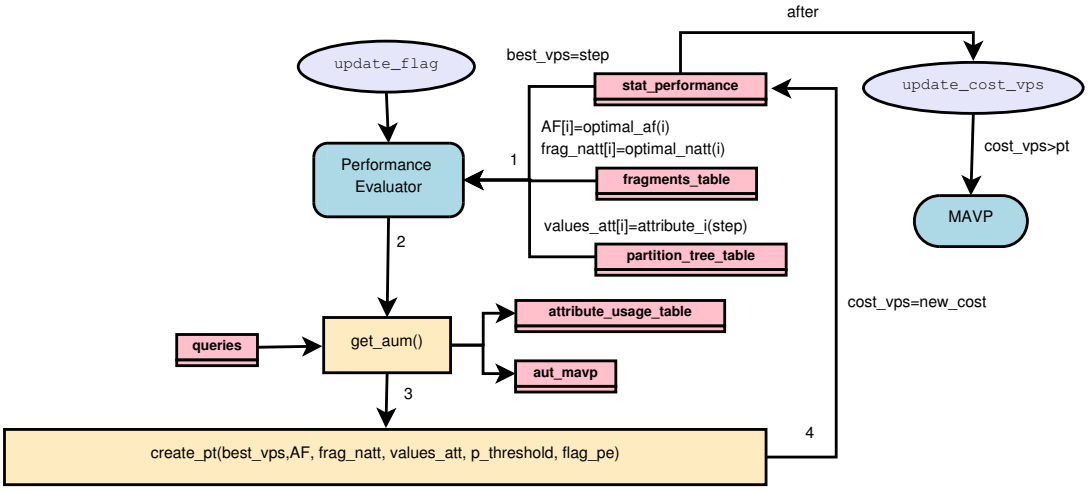


Figura 7.25: Proceso de evaluación de desempeño.

Después de actualizar *cost_ vps* en **stat_performance**, el disparador **update_cost_ vps** verifica si el nuevo costo es mayor que el umbral de desempeño (*cost_ vps>pt*), en tal caso dispara MAVP. Este proceso se muestra en la Figura 7.25.

7.2.6. Obtención de un nuevo EFV por MAVP

En la fragmentación vertical dinámica, MAVP es disparado por el *Evaluador de Desempeño* (*update_ cost_ vps*). El disparador **update_cost_vps** obtiene la información que MAVP necesita como entrada para obtener un EFV. Además elimina los datos de **partition_tree_table** y **fragments_table**. Por último dispara MAVP, el cual realiza los mismos pasos que en la fragmentación vertical estática (Sección 7.1.3). Este proceso se muestra en la Figura 7.26.

Antes de que MAVP actualice *best_ cost_ vps* en la tabla **stat_performance**, se dispara el trigger **update_best_cost_vps**, el cual realiza lo siguiente:

1. Actualiza los valores *previousna* y *previousnq* en **stat_attributes** y **stat_queries** respectivamente, a *previousna* le asigna el número de atributos, y a *previousnq* le asigna el número de consultas.

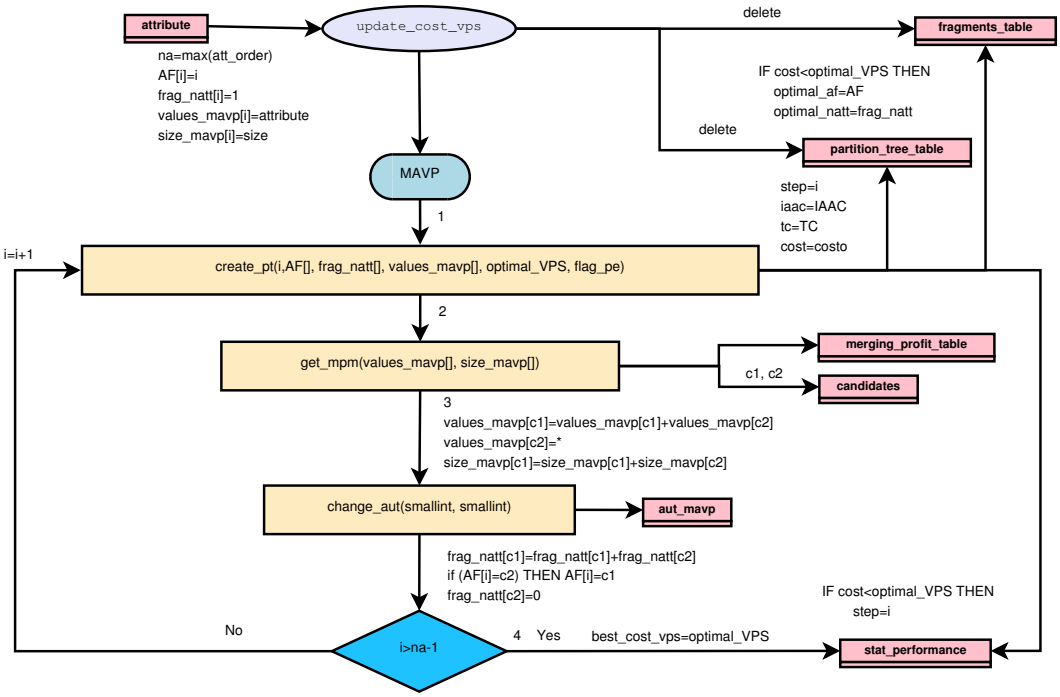


Figura 7.26: Proceso de obtención del EFV en la fragmentación dinámica.

2. Actualiza *prev_freq* de todas las consultas en la tabla **queries**, asignándole el valor de *frequency* y a *frequency* le asigna un cero.

3. Actualiza *prev_size* de todos los atributos en la tabla **attribute**, asignándole el valor de *size*.

4. Verifica si es la primera ejecución de MAVP, como no es, entonces actualiza los cambios actuales a cero (*currentnq, currentna, currentchanges, currentchangea, currentchangeq, currentchangef*) y a los cambios previos les asigna el promedio de los cambios anteriores y los cambios nuevos (*previouschanges, previouschangea, previouschangeq, previouschangef*), finalmente cabia *flag* a verdadera en **stat_performance.**

5. Antes de actualizar *flag* en **stat_performance**, el disparador **update_flag_pt** verifica si *flag=true*, entonces obtiene la suma (*sum*) de los cambios mayores a cero,

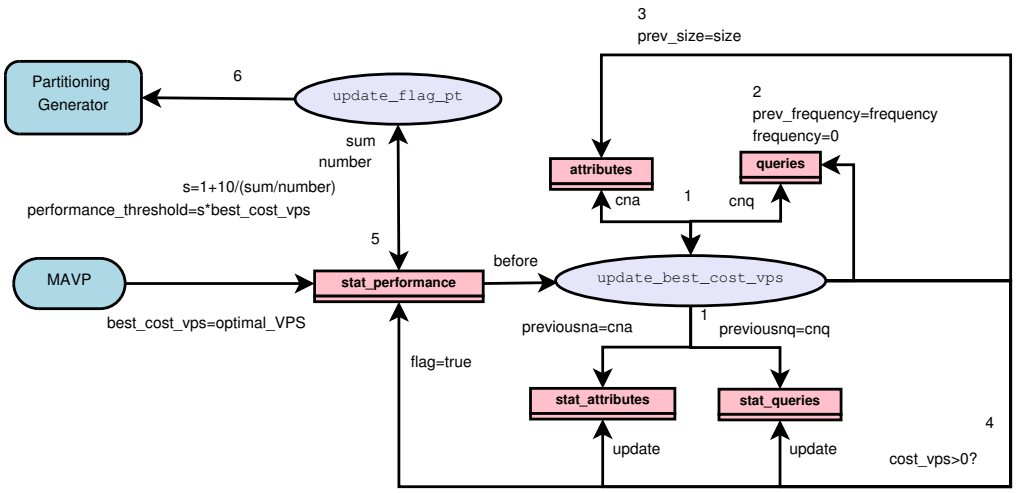


Figura 7.27: Proceso de cambio de estadísticas al obtener un EFV en la fragmentación vertical dinámica.

si esta suma es mayor a cero, la divide entre el número de cambios mayores a cero (*number*) y el resultado lo divide entre 10 para permitir un margen razonable de degradación de desempeño y le agrega 1 para que el umbral de desempeño (*performance_threshold*) siempre sea mayor o igual que el costo del EFV óptimo, este resultado es la variable de cambio de la base de datos multimedia (s). Después multiplica la variable de cambio s por el costo del EFV óptimo (*best_cost_vps*) y almacena los valores de la suma (*sum*) del número de cambios mayores a cero (*number*), de la variable de cambio (s) y del umbral de desempeño (*performance_threshold*) en **stat_performance**.

6. Finalmente, **update_flag_pt** dispara el *Generador de Fragmentos*.

Este proceso se muestra en la Figura 7.27.

7.2.7. Materialización del nuevo EFV

En la fragmentación vertical dinámica el *Generador de Fragmentos* realiza los mismos pasos que en la fragmentación vertical dinámica. Estos pasos se describieron en la sección 6.1.4. La única diferencia es que el *Generador de Fragmentos* es disparado por

update_flag_pt en la fragmentación vertical dinámica (figura 6.27), mientras que en la fragmentación vertical estática es el disparador **after_update_att** quien se encarga de invocar el *Generador de Fragmentos* (figura 6.8).

7.3. Descripción del esquema DYMOND inicial

En esta sección se describen las tablas, disparadores y funciones necesarias para iniciar el proceso de fragmentación vertical.

Tabla postgres_log. Almacena las consultas que se encuentran en el log de transacciones.

Disparadores:

- **current_database**. Antes de insertar en la tabla **postgres_log,** verifica que sólo se inserten las consultas realizadas en la base de datos en la que se ejecuta el script *dymond.sql* (que es la base de datos donde se encuentra la tabla que se desea fragmentar).

- **delete_postgreslog**. Después de eliminar en la tabla **postgres_log**, dispara las funciones *compare_att()* y *copy_log()*. Este disparador se ejecuta por cada sentencia de eliminación, esto significa que si se eliminan todas las filas de la tabla **postgres_log**, este disparador sólo se ejecutará una vez.

- **sc_getqueries**. Después de insertar en la tabla **postgres_log**, si no es la primera ejecución de DYMOND, entonces dispara el recolector de estadísticas. Este disparador se ejecuta por cada sentencia de inserción, por lo que si se insertan todas las consultas del log de transacciones, este disparador sólo se ejecutará una vez.

Tabla queries. Contiene información (id, descripción, frecuencia anterior, frecuencia actual) sobre las consultas que utilizan atributos de la tabla que se desea fragmentar.

Disparadores:

- **insert_queries.** Antes de insertar en la tabla **queries**, verifica si la consulta utiliza los mismos atributos de alguna consulta que ya esta almacenada en tal caso sólo se incrementa su frecuencia en 1.

- **delete_queries**. Después de eliminar en la tabla **queries**, cambia el identificador de las consultas con el fin de mantener la integridad de la tabla.

- **after_id_queries (Regla 8)**. Después de insertar o eliminar en la tabla **queries**, si no es la fragmentación inicial, entonces actualiza *currentnq* (el número de consultas insertadas o eliminadas) en la tabla **stat_queries**.

- **after_insert_queries**. Después de insertar una consulta en la tabla **queries**, si el número de consultas almacenadas en **queries** es mayor al identificador (*id*) máximo, entonces a dicho *id* se le asigna el número total de consultas, esto se hace con el objetivo de que las consultas tengan un *id* consecutivo.

Funciones:

1. *copy_log().* Copia las consultas del log de transacciones a la tabla **postgres_log**.

2. *get_queries(text).* Si es la fragmentación inicial, entonces crea las tablas y disparadores necesarios para continuar el proceso de fragmentación vertical y almacena las consultas ejecutadas en la tabla que se desea fragmentar (que se encuentran en la tabla **postgres_log**) en la tabla **queries**.

3. *get_aum().* Obtiene la tabla **attribute_usage_table** de la tabla **queries**.

4. *compare_att().* Detecta los cambios ocurridos en los atributos de la tabla fragmentada verticalmente.

5. *get_mpm(text[], bigint[]).* Obtiene la **merging_profit_table**.

6. *change_aut(smallint, smallint).* Cambia la **aut_table** durante la ejecución de MAVP.

7. *create_pt(smallint, smallint[], smallint[], text[], numeric, boolean).* Obtiene el costo de un EFV.

7.4. Descripción de las tablas y disparadores finales

Tabla attribute. Almacena información (id, nombre, tamaño anterior, tamaño actual) sobre los atributos de la tabla que se desea fragmentar.

Disparadores:

- **id_att** (**Regla 1**). Antes de insertar o eliminar en la tabla **attribute**, actualiza *currentna* (el número de atributos insertados o eliminados) en la tabla **stat_attributes** y modifica las tablas **attribute_usage_table, aut_table, y merging_profit_table** para que sean consistentes con la tabla **queries**.

- **update_att (Regla 4)**. Antes de actualizar en la tabla **attribute**, entonces:

  1. Si se actualiza el nombre de un atributo, entonces también se actualiza el nombre de dicho atributo en las tablas **attribute_usage_table, aut_table, y merging_profit_table**.

  2. Si se actualiza el tamaño de un atributo, entonces:

    2.1. Si el valor actualizado es positivo y no es la primera fragmentación, entonces se actualiza el valor *currenchanges* (el porcentaje de cambio en el tamaño de los atributos) en la tabla **stat_attributes**.

    2.2. Si el valor actualizado es igual a -5, quiere decir que el atributo es multimedia, y en tal caso, el valor no se actualiza.

    2.3. Si el valor actualizado es cero, entonces se pide al DBA que ingrese un valor positivo para el tamaño del atributo.

- **after_update_att**. Después de actualizar el tamaño de un atributo multimedia, si el nuevo tamaño es positivo y es la primera fragmentación, entonces verifica que

el DBA ya haya ingresado todos los tamaños de los atributos, en tal caso dispara el algoritmo MAVP.

Tabla stat_attributes. Almacena las estadísticas relacionadas con los atributos.

Disparadores:

- **update_currentna** (**Regla 2**). Antes de actualizar *currentna* en la tabla **stat_attributes**, si no es la primera fragmentación y el valor actualizado es mayor a cero, entonces se actualiza *currentchangea* (el porcentaje de cambio en el número de atributos) en la tabla **stat_attributes**.

- *update_currentchangea* (**Regla 3**). Después de actualizar *currentchangea* en la tabla **stat_attributes**, si el porcentaje de cambio en el número de atributos actual es mayor que el porcentaje previo (*currentchangea*>*previouschangea*), entonces *flag_ca* (la bandera que indica el disparo del evaluador de desempeño debido a cambios suficientes en el número de atributos) se vuelve verdadera en la tabla **stat_flags**.

- *update_currentchanges* (**Regla 5 y 6**). Antes de actualizar *currentchanges* en la tabla **stat_attributes**, si el porcentaje de cambio en el tamaño de los atributos (*currentchanges*) es menor a cero, entonces *currentchanges* se multiplica por -1 y si el valor actualizado es menor al valor anterior (new.*currentchanges*<old.*currentchanges*) entonces se deja el valor anterior.

- *after_update_currentchanges* (**Regla 7**). Después de actualizar *currentchanges* en la tabla **stat_attributes**, si el porcentaje de cambio en el tamaños de los atributos es mayor que el porcentaje previo (*currentchanges*>*previouschanges*), entonces *flag_cs* (la bandera que indica el disparo del evaluador de desempeño debido a cambios suficientes en el tamaño de los atributos) se vuelve verdadera en la tabla **stat_flags**.

Tabla stat_queries. Almacena las estadísticas relacionadas con las consultas.

Disparadores:

- *update_ currentnq* (**Regla 9**). Antes de actualizar *currentnq* en la tabla **stat_ queries**, si no es la primera fragmentación y el valor actualizado es mayor a cero, entonces se actualiza *currentchangeq* (el porcentaje de cambio en el número de consultas) en la tabla **stat_ queries**.

- *update_ currentchangeq* (**Regla 10**). Después de actualizar *currentchangeq* en la tabla **stat_ queries**, si el porcentaje de cambio en el número de consultas actual es mayor que el porcentaje previo (*currentchangeq*>*previouschangeq*), entonces *flag_ cq* (la bandera que indica el disparo del evaluador de desempeño debido a cambios suficientes en el número de consultas) se vuelve verdadera en la tabla **stat_ flags**.

- *update_ currentchangef* (**Reglas 12 y 13**). Antes de actualizar *currentchangef* en la tabla **stat_ queries**, si el porcentaje de cambio en la frecuencia de las consultas (*currentchangef*) es menor a cero, entonces *currentchangef* se multiplica por -1 y si el valor actualizado es menor al valor anterior (new.*currentchangef*<old.*currentchangef*) entonces se deja el valor anterior.

- *after_ update_ currentchangef* (**Regla 14**). Después de actualiza *currentchangef* en la tabla **stat_ queries**, si el porcentaje de cambio en la frecuencia de las consultas es mayor que el porcentaje previo (*currentchangef*>*previouschangef*), entonces *flag_ cf* (la bandera que indica el disparo del evaluador de desempeño debido a cambios suficientes en la frecuencia de las consultas) se vuelve verdadera en la tabla **stat_ flags**.

Tabla stat_ flags. Contiene las estadísticas relacionadas con las banderas para disparar el evaluador de desempeño.

Disparadores:

- *update_ flag (***Regla 15***).* Después de actualizar en la tabla **stat_ flags**, si las banderas que indican que ya se revisaron los cambios en los atributos (*flag_ att*), los cambios en el número de consultas (*flag_ queries*) y los cambios en la frecuencia de

las consultas (*flag_frequency*) son verdaderas, entonces si cualquiera de las banderas *flag_ca*, *flag_cq*, *flag_cs*, *flag_cf* es verdadera, se dispara el evaluador de desempeño.

Tabla attribute_usage_table. Almacena los atributos que se utilizan en cada consulta.

Disparadores:

- *delete_queries_aum*. Después de eliminar en **attribute_usage_table**, cambia el identificador de las consultas con el fin de mantener la integridad de la tabla.

Tabla aut_mavp. Es una copia de la **attribute_usage_table** utilizada por el algoritmo MAVP para encontrar un EFV. La diferencia entre la **aut_mavp** y la **attribute_usage_table** es que la primera cambia durante la ejecución de MAVP mientras que la segunda permanece igual durante la ejecución de MAVP.

Disparadores:

- *delete_aut_mavp*. Después de eliminar en **aut_mavp**, cambia el identificador de las consultas con el fin de mantener la integridad de la tabla.

Tabla stat_performance. Contiene las estadísticas relacionadas con el desempeño de la base de datos multimedia.

Disparadores:

- *update_cost_vps* (**Regla 17**): Después de actualizar el costo de un EFV (*cost_vps*) en **stat_performance**, si el costo es mayor a cero y es mayor al umbral de desempeño (*performance threshold*), entonces se ejecuta MAVP.

- *update_best_cost_vps* (**Regla 19**): Antes de actualizar el costo de un nuevo EFV (*best_cost_vps*) en **stat_performance**, se realizan cambios en las estadísticas. Si es la primera fragmentación, entonces a *performance_threshold* se le asigna el valor de *best_cost_vps*. En caso contrario, continúa con la modificación de las estadísticas y actualiza *flag* en **stat_performance** a verdadera.

- *update_flag_pt* (**Regla 20**): Antes de actualizar *flag* en **stat_performance**, si *flag* es verdadera, entonces obtiene el umbral de desempeño.

7.5. Conclusión

En esta sección se describió el desarrollo del sistema activo de fragmentación vertical dinámica para bases de datos multimedia DYMOND en el DBMS PostgreSQL, las reglas activas presentadas en el Capítulo 6 se implementaron como disparadores o triggers.

Capítulo 8

Caso de estudio

Se demostrará el funcionamiento de DYMOND a través de un caso de estudio de la empresa Servicios de Reparación de Maquinaria (SEREMAQ), la cual tiene una base de datos multimedia que almacena información de los equipos: motobombas, podadoras, desmalezadoras, motosierras, aspersoras, etc. Dicha información se almacena en la tabla **equipo** (*id, nombre, marca, modelo, caballaje, descripcion, imagen, grafico, animacion, audio, video*). Esta tabla tiene 100 tuplas y su definición en PostgreSQL es:

CREATE TABLE **equipo**

(

id character(5) NOT NULL,

nombre character(20),

marca character(20),

modelo character(15),

caballaje character(10),

descripcion character(250),

imagen bytea,

grafico bytea,

animacion bytea,

audio bytea,

video bytea,

CONSTRAINT *equipo_pkey* PRIMARY KEY (*id*)

)

A continuación se presenta el proceso de fragmentación vertical estática y dinámica de la tabla **equipo** dentro de la MMDB SEREMAQ.

8.1. Fragmentación vertical estática

Cuando el DBA ejecuta DYMOND por primera vez en la MMDB SEREMAQ se crea el esquema *dymond* como puede verse en la Figura 8.1.

Las consultas que se ejecutan en la tabla **equipo** son:

q_1: SELECT *id*, *nombre* FROM **equipo** WHERE *nombre*='MOTOBOMBA'

q_2: SELECT *imagen* FROM **equipo** WHERE *nombre*='MOTOBOMBA'

q_3: SELECT *imagen*, *grafico* FROM **equipo** WHERE *nombre*='MOTOSIERRA'

El *Recolector de Estadísticas* almacena estas consultas en la tabla **queries** que se muestra en la Figura 8.2. Como es la fragmentación vertical inicial, el *Recolector de Estadísticas* de DYMOND obtiene la tabla de uso de atributos (**attribute_usage_table**) de la Figura 8.3.

Posteriormente, se dispara MAVP que obtiene la tabla de fragmentación de la Figura 8.4 (**partition_tree_table).** MAVP almacena el menor costo en la tabla **stat_performance**, la cual se muestra en la Figura 8.5. En este caso el EFV con menor costo es el paso 7 (*id, nombre*) (*marca, modelo, caballaje, descripcion, animacion, audio, video*) (*imagen*) (*grafico*) con un costo de 30220. El costo del EFV se obtiene de la siguiente manera:

$IAAC(q_1) = 0, TC(q_1) = 0, Costo(q_1) = 0$

$IAAC(q_2) = frequency_2 * size_{id} = 3 * 5 = 15$

$TC(q_2) = frequency_2^2 * size_{nombre} = 3^2 * 20 = 180$

$Costo(q_2) = 15 + 180 = 195$

$IAAC(q_3) = frequency_3 * size_{id} = 1 * 5 = 5$

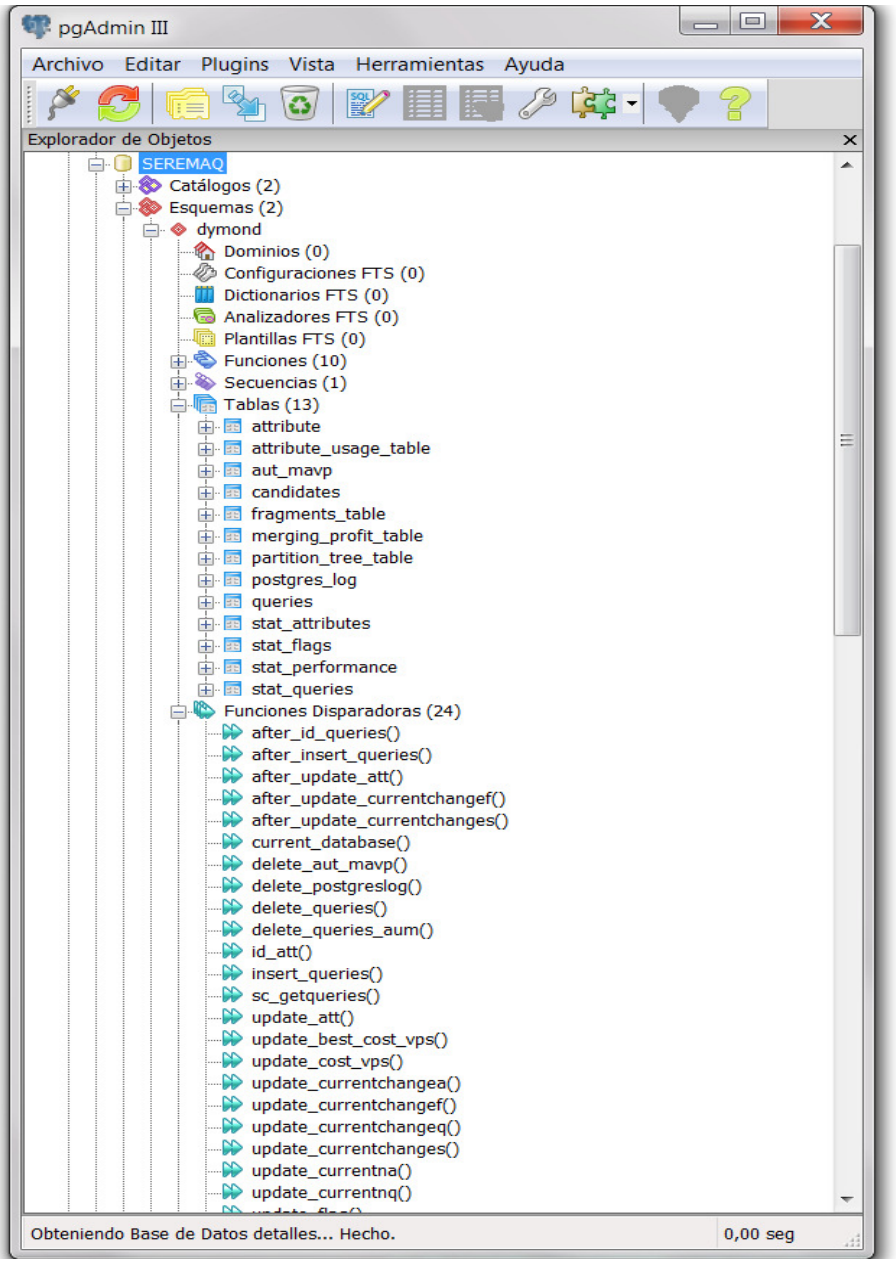


Figura 8.1: Esquema dymond inicial en SEREMAQ.

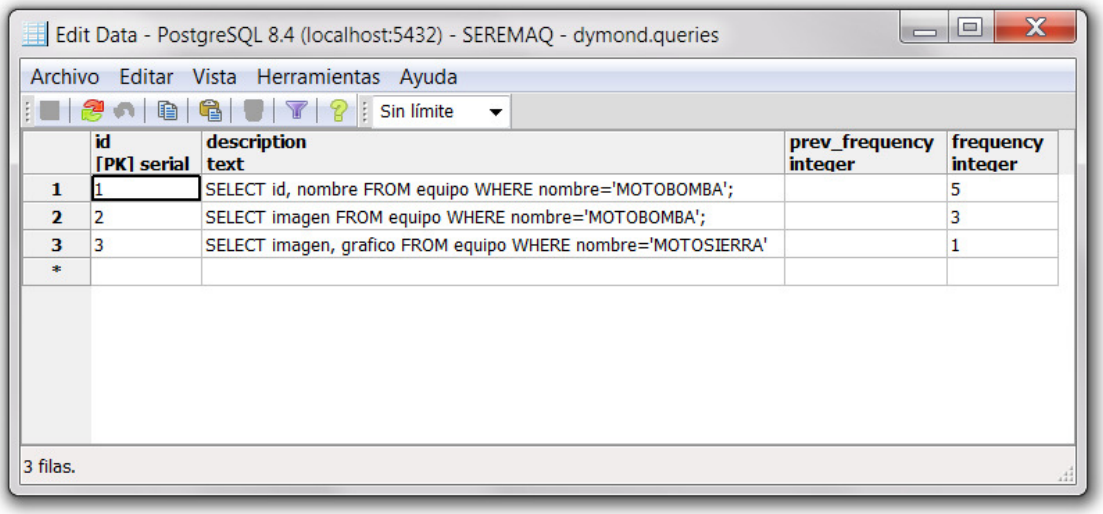


Figura 8.2: Tabla de consultas en la fragmentación vertical estática.

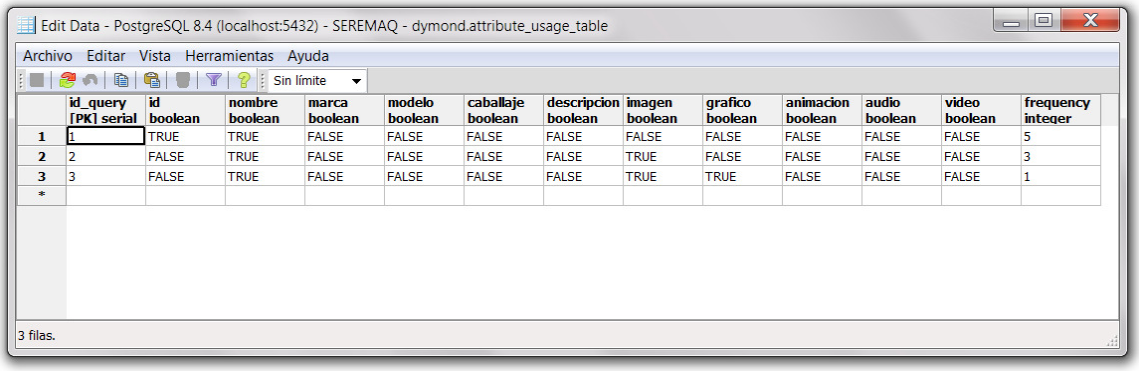
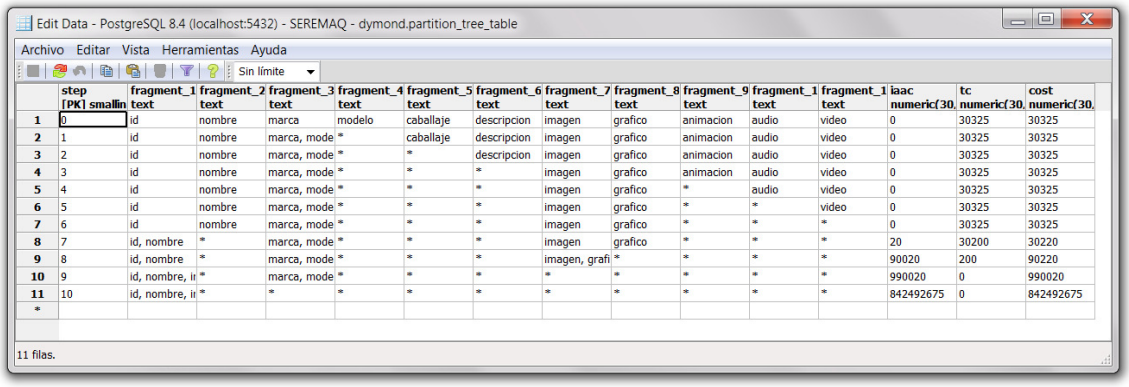


Figura 8.3: Tabla de uso de atributos.



	step [PK] smallin	fragment_1 text	fragment_2 text	fragment_3 text	fragment_4 text	fragment_5 text	fragment_6 text	fragment_7 text	fragment_8 text	fragment_9 text	fragment_1 text	fragment_1 text	iaac numeric(30,	tc numeric(30,	cost numeric(30,
1	0	id	nombre	marca	modelo	caballaje	descripcion	imagen	grafico	animacion	audio	video	0	30325	30325
2	1	id	nombre	marca, mode	*	caballaje	descripcion	imagen	grafico	animacion	audio	video	0	30325	30325
3	2	id	nombre	marca, mode	*	*	descripcion	imagen	grafico	animacion	audio	video	0	30325	30325
4	3	id	nombre	marca, mode	*	*	*	imagen	grafico	animacion	audio	video	0	30325	30325
5	4	id	nombre	marca, mode	*	*	*	imagen	grafico	*	audio	video	0	30325	30325
6	5	id	nombre	marca, mode	*	*	*	imagen	grafico	*	*	video	0	30325	30325
7	6	id	nombre	marca, mode	*	*	*	imagen	grafico	*	*	*	0	30325	30325
8	7	id, nombre	*	marca, mode	*	*	*	imagen	grafico	*	*	*	20	30200	30220
9	8	id, nombre	*	marca, mode	*	*	*	imagen, grafi	*	*	*	*	90020	200	90220
10	9	id, nombre, ir	*	marca, mode	*	*	*	*	*	*	*	*	990020	0	990020
11	10	id, nombre, ir	*	*	*	*	*	*	*	*	*	*	842492675	0	842492675
*															

Figura 8.4: Tabla de fragmentación de atributos.

$$TC(q_3) = frequency_3^2 * (size_{nombre} + size_{grafico)} = 1^2 * 30020 = 30020$$

$$Costo(q_3) = 5 + 30020 = 30025$$

$$CostoEFV = Costo(q_1) + Costo(q_2) + Costo(q_3) = 0 + 195 + 30025 = 30220$$

Finalmente, el *Generador de Fragmentos* materializa el EFV, esto es, crea los fragmentos como índices. Esto se muestra en la Figura 8.6.

8.1.1. Comparación de tiempo y costo de ejecución de las consultas

En la Tabla 8.1 se presenta el tiempo de respuesta de las consultas en milisegundos, cada consulta se ejecutó diez veces y se obtuvo el promedio del tiempo de ejecución. EL DBMS respondió las consultas más rápido utilizando los fragmentos (índices) generados por DYMOND (FVE) en SEREMAQ, esto se debe a que en la MMDB SEREMAQ no fragmentada (NF), las consultas se respondieron por medio de búsquedas secuenciales, por lo que fue necesario recorrer toda la tabla hasta encontrar los datos deseados.

La mayoría de las consultas sólo hacen referencia a una pequeña parte de los atributos y registros del sistema, por ejemplo, la consulta q_1 sólo utiliza los atributos *id* y *nombre*. Los índices son estructuras que ayudan a localizar rápidamente los registros deseados de una tabla, sin examinar todos los registros [2]. Para recuperar los equipos cuyo nombre

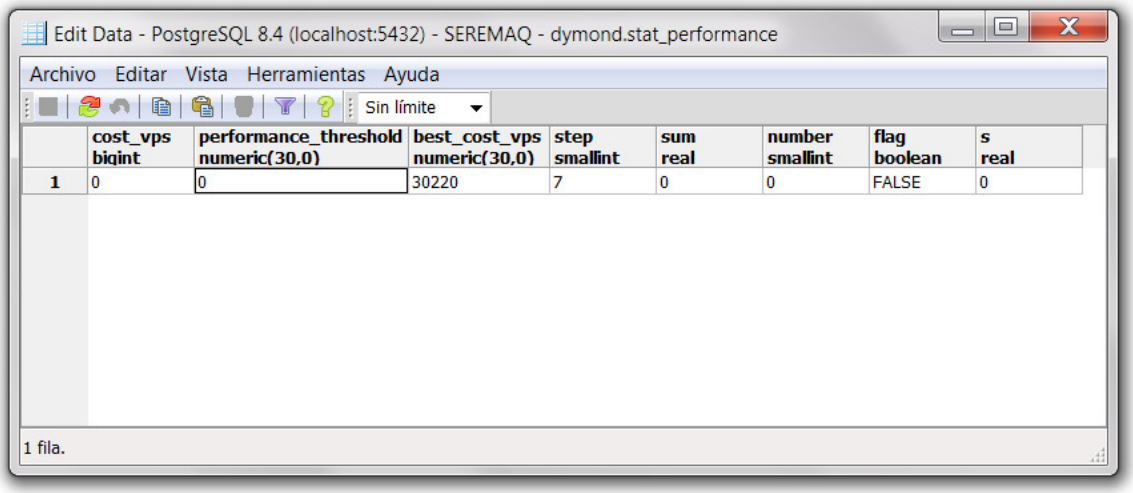


Figura 8.5: Tabla de estadísticas de desempeño de la MMDB.

es igual a 'MOTOBOMBA', el DBMS busca en el índice formado por los atributos (*id*, *nombre*) los bloques de disco en que se encuentra el registro correspondiente, y entonces sólo extrae ese bloque de disco para obtener los datos requeridos.

La comparación de costo de ejecución de las consultas en la MMDB SEREMAQ no fragmentada e implementando el EFV inicial generado por DYMOND se muestran en la Tabla 8.3 y en la Tabla 8.2 se encuentran los EFVs. El modelo de costo propuesto en esta tesis y publicado en [116] se utilizó como base para evaluar ambos esquemas. Como puede verse el costo de acceso a atributos irrelevantes (IAAC) se incrementa considerablemente en SEREMAQ no fragmentada, mientras que el costo de transporte (TC) se reduce. La razón de esto es que todos los atributos se encuentran en un sólo fragmento y las consultas deben cargar en memoria todos los atributos presentes en el fragmento aunque sólo requieran unos cuantos, en el caso de los atributos multimedia que son de gran tamaño, debe evitarse el acceso a atributos irrelevantes para reducir considerablemente el costo de ejecución de las consultas, esto se logra en el EFV creado por DYMOND. Aunque el costo de transporte se incrementa en el EFV inicial generado por DYMOND, el ahorro de costo de ejecución es muy significativo.

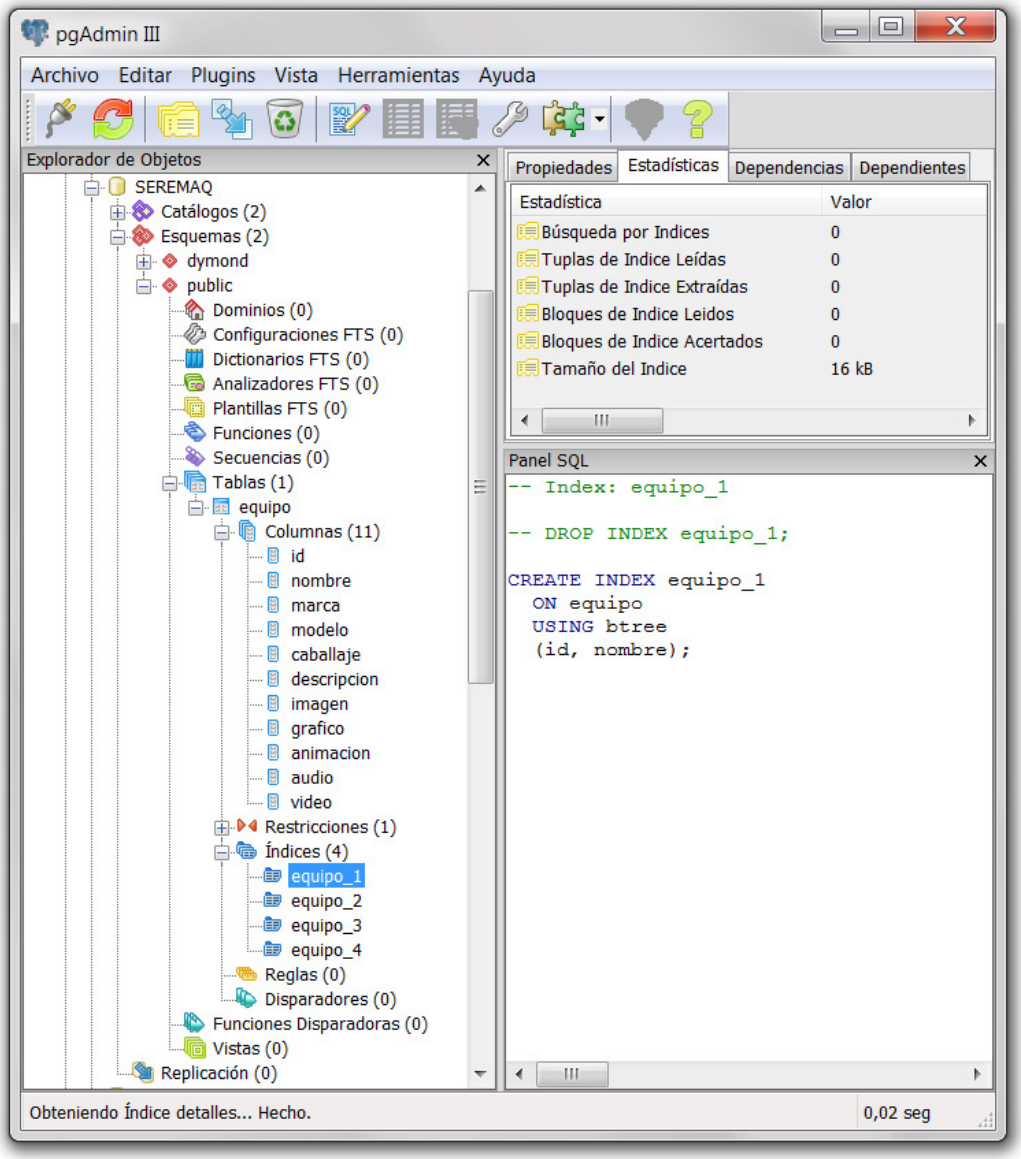


Figura 8.6: Índices creados por el generador de fragmentos.

Tabla 8.1: Tiempo de respuesta de consultas.

Consulta	NF	FVE
q_1	21	12
q_2	2639	2539
q_3	2692	2691

Tabla 8.2: Comparación de EFVs.

SEREMAQ	EFV
NF	*(id, nombre, marca, modelo, caballaje, descripcion, imagen, grafico, animacion, audio, video)*
FVE	*(id, nombre)* *(marca, modelo, caballaje, descripcion, animacion, audio, video)* *(imagen) (grafico)*

8.2. Fragmentación vertical dinámica

Si se agrega el atributo *precio* a la tabla equipo, se realizan algunos cambios en la frecuencia de las consultas y se insertan dos nuevas consultas:

`q4:SELECT` *nombre*`,` *audio*`,` *video* `FROM` **equipo** `WHERE` *id*`='MB1'`

`q5:SELECT` *id, animacion, grafico* `FROM` **equipo** `WHERE precio<3000`

Como el DBA agregó el atributo precio a la tabla **equipo**, este se agrega automáticamente a la vista *attributes* de la Figura 8.7.

La función *compare_ att()* detecta el nuevo atributo en la vista *attributes* y lo inserta en la tabla **attribute** que se presenta en la Figura 8.8.

Como se puede ver en la Figura 8.9, ahora la tabla **queries** contiene las tres primeras consultas, la frecuencia de la consulta q_1 cambió de 5 a 6, q_2 de 3 a 5, la frecuencia de q_3 no

Tabla 8.3: Comparación de costo de ejecución de consultas.

query	NF			FVE		
	IAAC	TC	Costo	TC	IAAC	Costo
q1	468401475	0	468401475	0	0	0
q2	280590900	0	280590900	15	180	195
q3	93500300	0	93500300	5	30020	30025

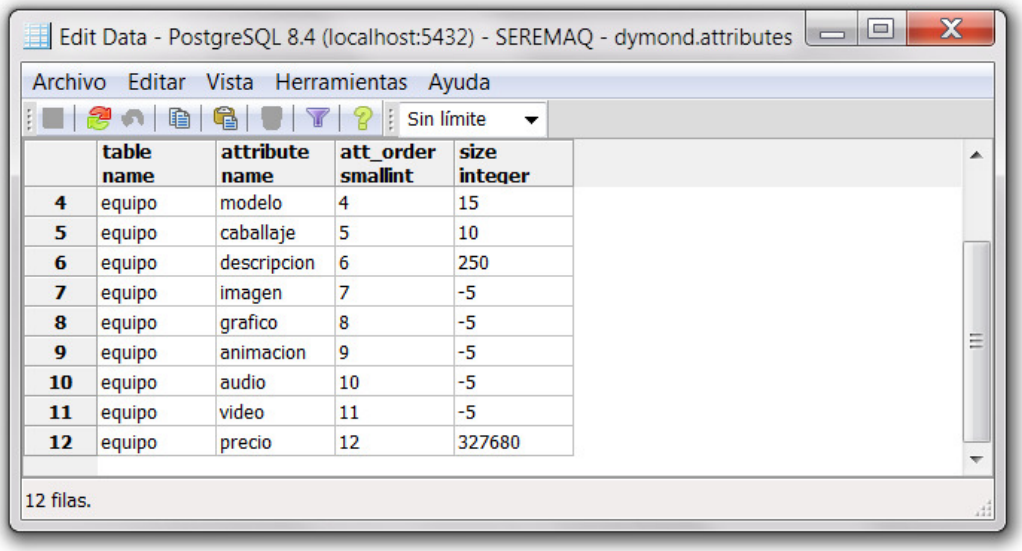


	table name	attribute name	att_order smallint	size integer
4	equipo	modelo	4	15
5	equipo	caballaje	5	10
6	equipo	descripcion	6	250
7	equipo	imagen	7	-5
8	equipo	grafico	8	-5
9	equipo	animacion	9	-5
10	equipo	audio	10	-5
11	equipo	video	11	-5
12	equipo	precio	12	327680

Figura 8.7: Vista de los atributos de la tabla equipo.

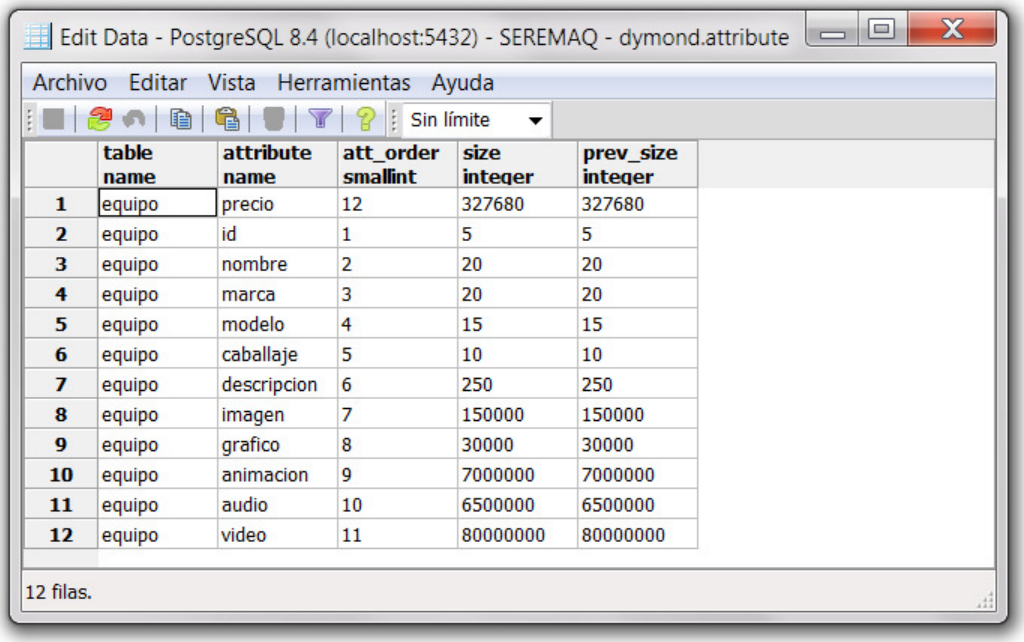
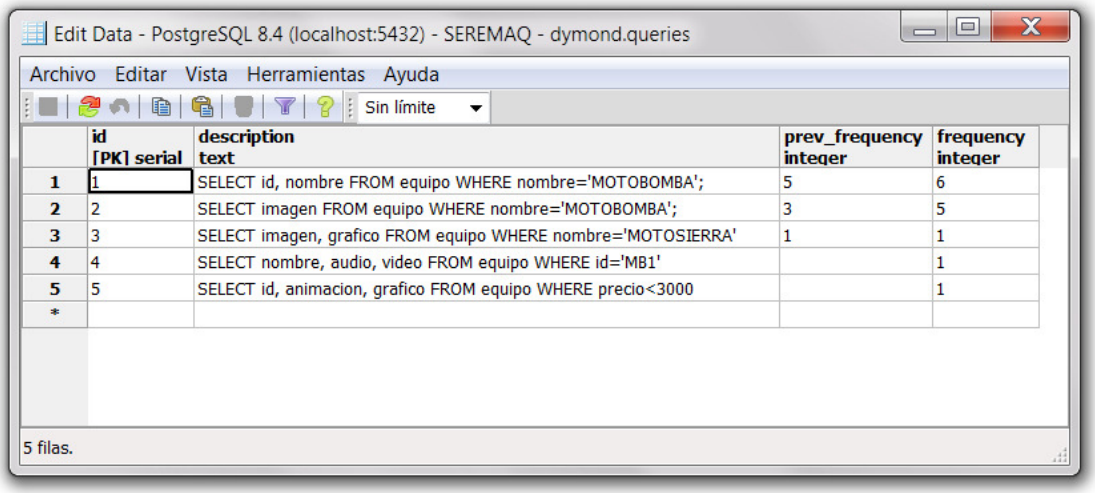


	table name	attribute name	att_order smallint	size integer	prev_size integer
1	equipo	precio	12	327680	327680
2	equipo	id	1	5	5
3	equipo	nombre	2	20	20
4	equipo	marca	3	20	20
5	equipo	modelo	4	15	15
6	equipo	caballaje	5	10	10
7	equipo	descripcion	6	250	250
8	equipo	imagen	7	150000	150000
9	equipo	grafico	8	30000	30000
10	equipo	animacion	9	7000000	7000000
11	equipo	audio	10	6500000	6500000
12	equipo	video	11	80000000	80000000

Figura 8.8: Tabla de atributos.



	id [PK] serial	description text	prev_frequency integer	frequency integer
1	1	SELECT id, nombre FROM equipo WHERE nombre='MOTOBOMBA';	5	6
2	2	SELECT imagen FROM equipo WHERE nombre='MOTOBOMBA';	3	5
3	3	SELECT imagen, grafico FROM equipo WHERE nombre='MOTOSIERRA'	1	1
4	4	SELECT nombre, audio, video FROM equipo WHERE id='MB1'		1
5	5	SELECT id, animacion, grafico FROM equipo WHERE precio<3000		1
*				

Figura 8.9: Tabla de consultas en la fragmentación vertical dinámica

cambió, pero también contiene dos nuevas consultas: q_4 que utiliza los atributos *nombre*, *audio*, *video*, *id* y q_5 que utiliza los atributos *id*, *animacion*, *grafico*, *precio*.

Como cambió el número de atributos, el número de consultas y la frecuencia de las consultas, las banderas *flag_ attributes*, *flag_ queries*, *flag frequency*, *flag_ ca*, *flag_ cq* y *flag_ cf* son verdaderas en **stat_performance**. Esto puede verse en la Figura 8.10.

Como las banderas anteriores son verdaderas, el *Recolector de Estadísticas* dispara del *Evaluador de Desempeño* que obtiene la nueva tabla de uso de atributos (**attribute_usage_table).** Dicha tabla se muestra en la Figura 8.11.

Posteriormente, el *Evaluador de Desempeño* calcula el nuevo costo del EFV actual y lo almacena en la tabla de estadísticas de desempeño (**stat_performance)**. Esto se demuestra en la Figura 8.12.

EL costo del EFV actualmente implementado (*id, nombre*) (*marca, modelo, caballaje, descripcion, animacion, audio, video*) (*imagen*) (*grafico*) (*precio*) es 93,888,870 porque:

IAAC(q_1)=0, TC(q_1)=0

IAAC(q_2)=25, TC(q_2)=500

IAAC(q_3)=5, TC(q_3)=30020

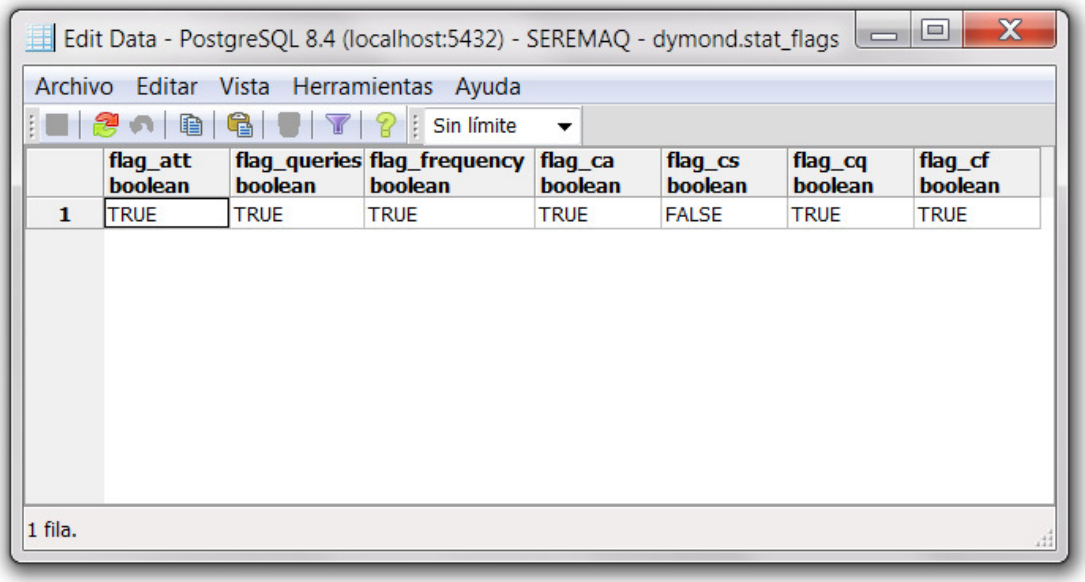


Figura 8.10: Tabla de banderas del recolector de estadísticas.

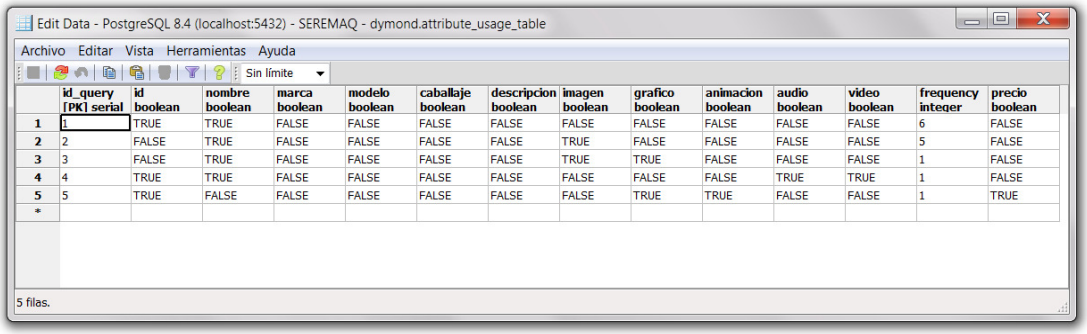


Figura 8.11: Nueva tabla de uso de atributos.

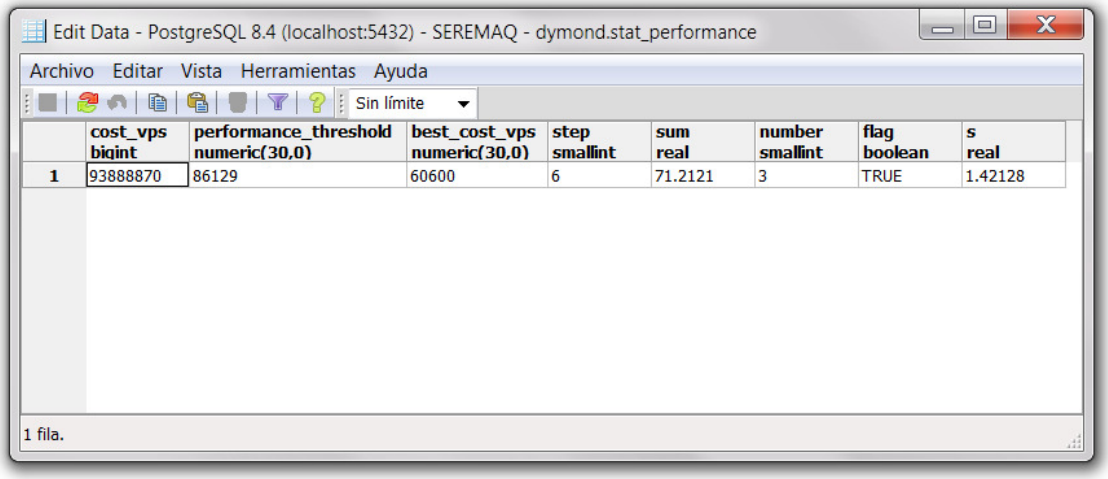


Figura 8.12: Tabla de desempeño de la MMDB en la fragmentación vertical dinámica.

IAAC(q_4)=7000295, TC(q_4)=25

IAAC(q_5)=86500315, TC(q_5)=357685

Costo=25+500+5+30020+25+7000295+86500315+357685=938888870

Como este costo es mayor que el umbral de desempeño (que en este caso es 0, porque es después de una fragmentación inicial) se dispara MAVP, que obtiene una nueva tabla de fragmentación de atributos (**partition_tree_table),** la cual se muestra en la Figura 8.13.

Ahora el nuevo EFV es el paso 6 (*id, nombre*) (*marca, modelo, caballaje, descripcion*) (*imagen*) (*grafico*) (*animacion, precio*) (*audio, video*) con un costo de 60,600 porque:

IAAC(q_1)=0, TC(q_1)=0, costo(q_1)=0

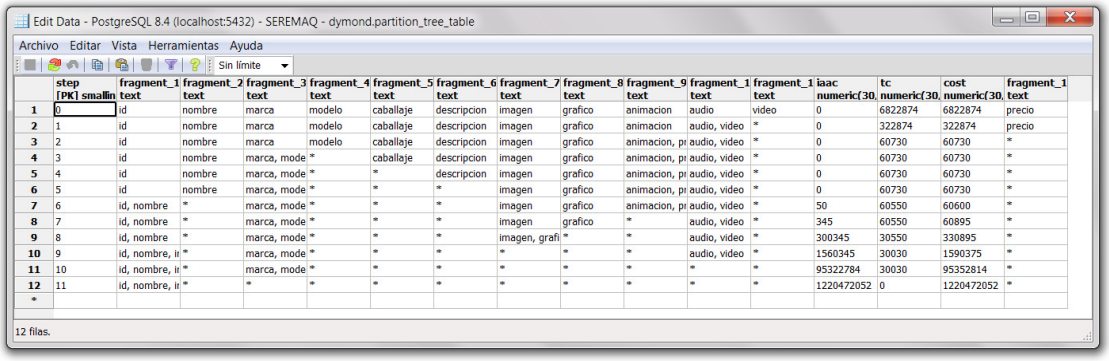
IAAC(q_2)=25, TC(q_2)=500, costo(q_2)=525

IAAC(q_3)=5, TC(q_3)=30,020, costo(q_3)=30,025

IAAC(q_4)=0, TC(q_4)=25, costo(q_4)=25

IAAC(q_5)=20, TC(q_5)=30,005, costo(q_5)=30,025

Costo=0+525+30,025+25+30,025=60,600 que se almacena en las estadísticas de desempeño de la MMDB (**stat_performance**).



	step [PK] smallin	fragment_1 text	fragment_2 text	fragment_3 text	fragment_4 text	fragment_5 text	fragment_6 text	fragment_7 text	fragment_8 text	fragment_9 text	fragment_1 text	fragment_1 text	iaac numeric(30,	tc numeric(30,	cost numeric(30,	fragment_1 text
1	0	id	nombre	marca	modelo	caballaje	descripcion	imagen	grafico	animacion	audio	video	0	6822874	6822874	precio
2	1	id	nombre	marca	modelo	caballaje	descripcion	imagen	grafico	animacion	audio, video	*	0	322874	322874	precio
3	2	id	nombre	marca	modelo	caballaje	descripcion	imagen	grafico	animacion, pi	audio, video	*	0	60730	60730	*
4	3	id	nombre	marca, mode	*	caballaje	descripcion	imagen	grafico	animacion, pi	audio, video	*	0	60730	60730	*
5	4	id	nombre	marca, mode	*	*	descripcion	imagen	grafico	animacion, pi	audio, video	*	0	60730	60730	*
6	5	id	nombre	marca, mode	*	*	*	imagen	grafico	animacion, pi	audio, video	*	0	60730	60730	*
7	6	id, nombre	*	marca, mode	*	*	*	imagen	grafico	animacion, pi	audio, video	*	50	60550	60600	*
8	7	id, nombre	*	marca, mode	*	*	*	imagen	grafico	*	audio, video	*	345	60550	60895	*
9	8	id, nombre	*	marca, mode	*	*	*	imagen, grafi	*	*	audio, video	*	300345	30550	330895	*
10	9	id, nombre, ir	*	marca, mode	*	*	*	*	*	*	audio, video	*	1560345	30030	1590375	*
11	10	id, nombre, ir	*	marca, mode	*	*	*	*	*	*	*	*	95322784	30030	95352814	*
12	11	id, nombre, ir	*	*	*	*	*	*	*	*	*	*	1220472052	0	1220472052	*
*																

Figura 8.13: Nueva tabla de fragmentación de atributos.

El disparador **update_best_cost_vps** actualiza las estadísticas en **stat_attributes**, **stat_queries** y cambia el valor de la bandera que indica que ya se puede obtener el valor del umbral de desempeño (*flag*) a verdadera en **stat_performance**, como se muestra en las Figuras 8.12, 8.14 y 8.15.

En **stat_attributes,** el disparador **update_best_cost_vps** actualiza los valores de *previousna* al que le va a asignar el número de atributos actual de la tabla **equipo** $previouschangea = \frac{currentchangea + previouschangea}{2}$, como $currentchangea = \frac{1}{11} * 100$=9.09 y *previouschangea*=0, entonces *previouschangea*=4.54545. En **stat_queries** asigna el número de consultas que hay en **queries** a *previousnq* y a *currentnq*=0.

Debido a que $previouschangeq = \frac{currentchangeq + previouschangeq}{2}$, *previouschangeq*=33.33, porque $currentchangeq = \frac{2}{3} * 100$=66.66 y *previouschangeq*=0.

También $previouschangef = \frac{currentchangef + previouschangef}{2}$, ya que $currentchangef(q_1) = \frac{(6-5)}{5} * 100 = 20$ y $currentchangef(q_2) = \frac{(5-3)}{3} * 100$=66.66, como siempre se va a guardar el valor máximo de cambio en la frecuencia de los atributos, *currentchangef*=66.66 y *previouschangef*=0, entonces *previouschangef*=33.33. Como *flag*=true, el disparador **update_flag_pt** obtiene la variable $s = 1 + \frac{10}{71{,}21/3}$=1.4212 y el umbral de desempeño *performance_threshold*=*best_cost_vps***s*=60600*1.4212=86129.

Debido a que el costo del nuevo EFV (*best_cost_vps)* es menor que el costo del EFV

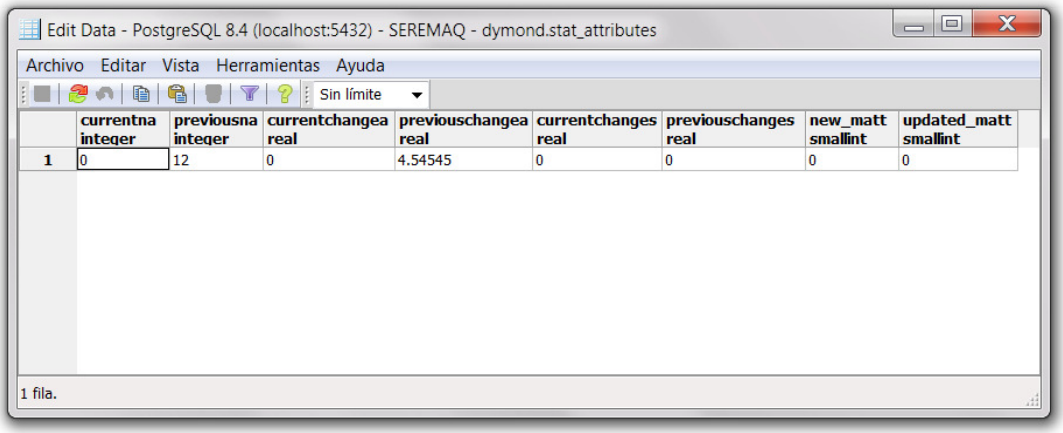


Figura 8.14: Tabla de estadísticas de los atributos.

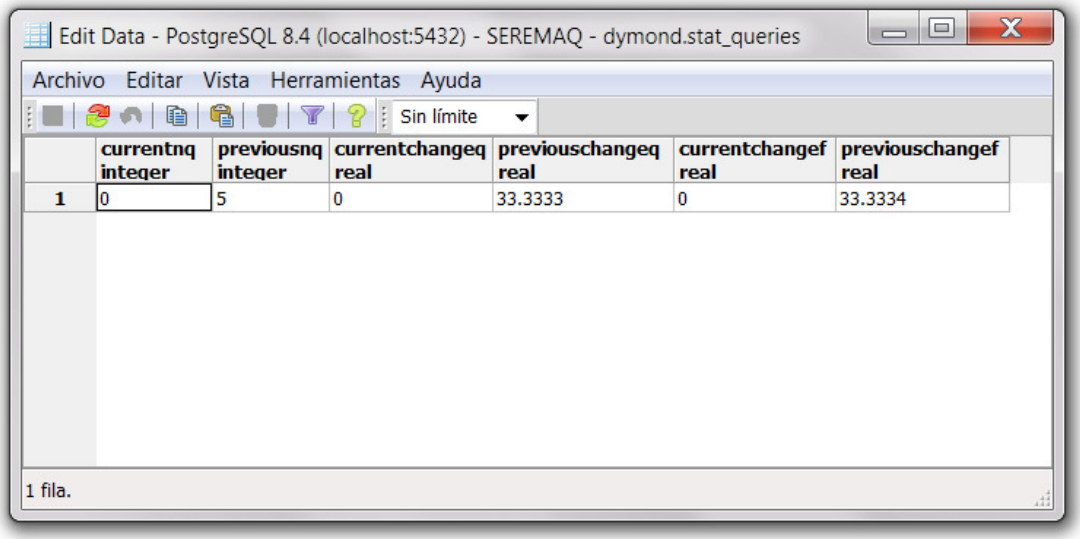


Figura 8.15: Tabla de estadísticas de las consultas.

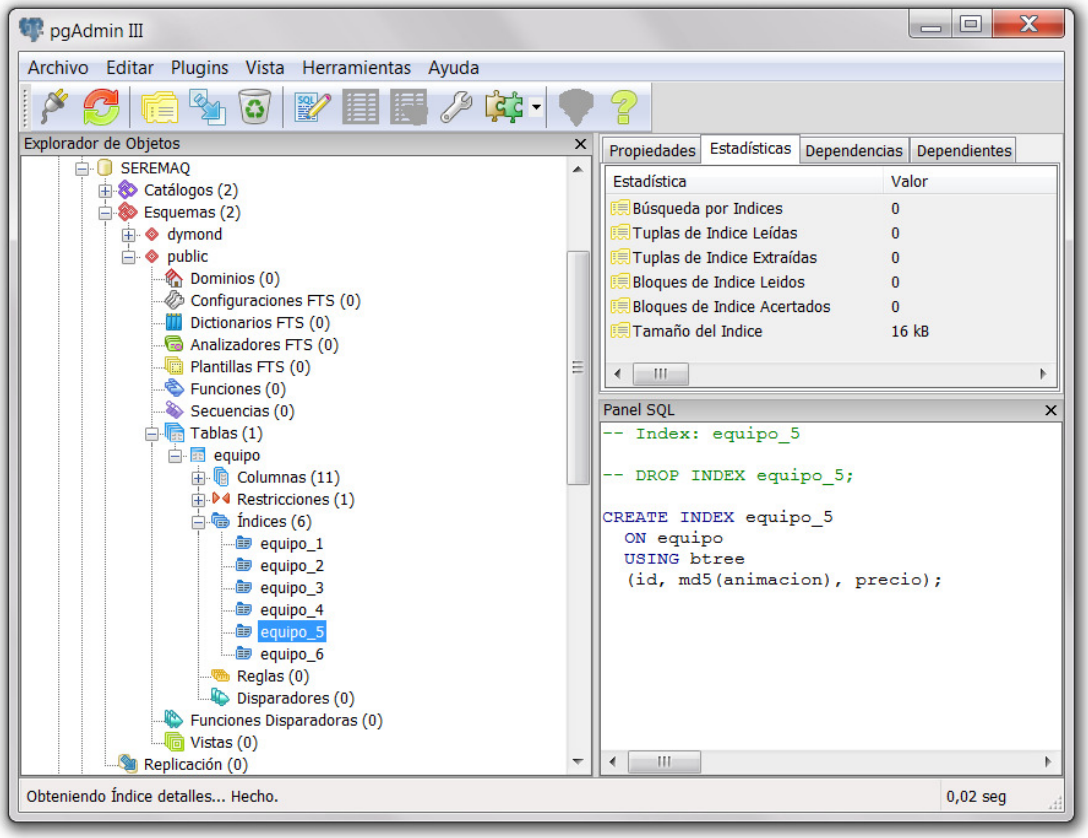


Figura 8.16: Nuevo EFV en SEREMAQ.

actualmente implementado (*cost_ vps)* entonces se dispara el *Generador de Fragmentos* que crea el nuevo EFV en SEREMAQ, éste se muestra en la Figura 8.16.

8.2.1. Comparación de tiempo y costo de ejecución de las consultas

Ahora en la Tabla 8.4 se compara el tiempo de respuesta de las consultas en SEREMAQ, en el primer caso se considera la MMDB no fragmentada (NF), en el segundo caso se ejecutaron las consultas cuando DYMOND implementa el EFV inicial (FVE) y, finalmente, se evalúo el tiempo de ejecucion utililizando el EFV obtenido por DYMOND dinámicamente usando reglas activas o disparadores (FVD), los EFVs se prensentan en la Tabla 8.5. Nuevamente el tiempo de respuesta es más bajo que en SEREMAQ no fragmentada. Sin

embargo, en las primeras tres consultas no hay diferencia entre FVE y FVD, la razón de esto es que en ambas se generó el mismo fragmento (*id, nombre*) y las consultas acceden a este fragmento nuevamente.

Como puede verse en la Tabla 8.6, la detección oportuna de los cambios en la MMDB le permiten a DYMOND crear un nuevo EFV que reduce en gran manera el costo de ejecución de las consultas. Las reglas activas le proporcionan a DYMOND la habilidad de reaccionar a cambios importantes tanto en los atributos como en las consultas, para evitar la degradación de desempeño del EFV, lo cual se traduce en un aumento en el tiempo de respuesta de las consultas. Por ejemplo, para responder la consulta q_5 en FVE el DBMS realiza una búsqueda secuencial debido a que no existe un fragmento adecuado a sus necesidades. Cuando DYMOND implementa el esquema FVD, crea el fragmento (*id, animacion, precio*) que es utilizado por la consulta q_5 obteniendo un menor tiempo de respuesta.

Tabla 8.4: Tiempo de respuesta de consultas.

Consulta	NF	FVE	FVD
q_1	21	12	12
q_2	2639	2539	2539
q_3	2692	2691	2691
q_4	11181	11164	11155
q_5	37690	37677	37669

Tabla 8.5: Comparación de EFVs.

SEREMAQ	EFV
NF	*(id, nombre, marca, modelo, caballaje, descripcion, imagen, grafico, animacion, audio, video)*
FVE	*(id, nombre)* *(marca, modelo, caballaje, descripcion, animacion, audio, video)* *(imagen) (grafico)(precio)*
FVD	(*id, nombre*) (*marca, modelo, caballaje, descripcion*)
	(*imagen*) (*grafico*) (*animacion, precio*) (*audio, video*)

Tabla 8.6: Comparación de costo de ejecución de consultas.

query	NF			FVE			FVD		
	IAAC	TC	Costo	TC	IAAC	Costo	TC	IAAC	Costo
q_1	468401475	0	468401475	0	0	0	0	0	0
q_2	280590900	0	280590900	25	500	525	25	500	525
q_3	93500300	0	93500300	5	30020	30025	5	30020	30025
q_4	7180295	0	7180295	7000295	25	7000320	0	25	25
q_5	86322635	0	86322635	86500315	357685	86858000	20	30005	30025

Capítulo 9

Conclusiones

En esta sección se presentan los resultados obtenidos en esta tesis y el trabajo futuro.

9.1. Conclusiones

El propósito de este trabajo de investigación es mejorar el tiempo de respuesta de las consultas en bases de datos multimedia (MMDB) distribuidas. La fragmentación vertical permite mejorar el desempeño de las consultas en estas bases de datos, ya que reduce la cantidad de atributos multimedia irrelevantes y atributos multimedia remotos accedidos por las consultas. En este trabajo se propone un algoritmo de fragmentación vertical para bases de datos multimedia distribuidas, llamado MAVP (Multimedia Adaptable Vertical Partitioning), que toma en cuenta los patrones de acceso y el tamaño de los atributos (obtenidos por un administrador de la base de datos en una fase de análisis previa) para generar un esquema de fragmentación vertical (EFV) que minimiza el costo de acceso a atributos irrelevantes y maximiza la localidad de la consultas, esto es, reduce el costo de acceso a atributos remotos en MMDBs distribuidas.

También se presenta un modelo de costo que permite evaluar distintos EFVs en MMDBs distribuidas. Este modelo mide el costo de acceso a attributos irrelevantes (IAAC), así como el costo de transporte o transferencia de atributos relevantes (TC). Para calcular IAAC,

el modelo de costo se basa en la frecuencia de acceso de las consultas, los atributos a los que acceden, así como en el tamaño y la ubicación de dichos atributos. Para obtener el TC, además de la información anterior, requiere conocer la ubicación de las consultas y el número de sitios de la MMDB distribuida.

Debido a que las MMDBs distribuidas son accedidas por muchos usuarios simultáneamente, los patrones de acceso cambian constantemente, como consecuencia es necesario un sistema de fragmentación vertical dinámica para lograr un desempeño de la base de datos eficiente. Por tal motivo se propone un sistema de fragmentación vertical dinámica para bases de datos relacionales llamado DYVEP y uno para MMDBs llamado DYMOND. Estos sistemas se basan en reglas activas que permiten detectar oportunamente cambios en los patrones de acceso, evaluar los cambios y disparar el proceso de fragmentación cuando ocurran suficientes cambios.

DYVEP implementa un algoritmo clásico de fragmentación vertical, por lo tanto, no toma en cuenta el tamaño de los atributos y no es adecuado para emplearse en MMDBs donde los atributos tienen tamaños muy variados. Por ejemplo, en una MMDBs se pueden tener videos que ocupen 1.5 GB de almacenamiento y atributos de tipo booleano que sólo ocupen 1 byte. Es preferible que una consulta acceda a un atributo irrelevante de tipo booleano que a un video, de igual forma, si la consulta requiere los dos atributos pero están en distintos sitios, es mejor que acceda remotamente al atributo booleano que al video. Así que el tamaño de los atributos es un factor muy importante a considerar en el proceso de fragmentación vertical en la MMDBSs.

La evaluación de DYVEP se realizó en el benchmark TPC-H, el cual es ampliamente utilizado para medir el desempeño de los DBMSs relacionales. El DBMS utilizado por DYVEP fue PostgreSQL debido a que es gratuito y de código libre. Las reglas activas se implementaron como disparadores por las siguientes razones:

- La base de datos almacena disparadores como si fueran datos normales, por lo que son persistentes y accesibles para todas las operaciones de la base de datos.

- Una vez se almacena un disparador en la base de datos, el sistema de base de datos asume la responsabilidad de ejecutarlo cada vez que ocurra el evento específicado y se satisfaga la condición correspondiente.

Se demostró que DYVEP permite reducir el tiempo de respuesta de las consultas en bases de datos relacionales. Sin embargo, DYVEP también tiene desventajas:

- Realiza todo el proceso de fragmentación en cada ejecución. Es decir, cada vez que se ejecuta DYVEP (por ejemplo, cada día) recolecta estadísticas, obtiene un nuevo EFV y materializa el EFV, aunque la tabla contenga las mismas consultas con la misma frecuencia.

- No considera los cambios en los atributos, por lo que si se eliminan, renombran o agregan atributos a la tabla fragmentada dinámicamente, el DBA debe eliminar el esquema *dyvep* de la base de datos y volver a crearlo para que se vuelvan a generar las tablas necesarias en el proceso de fragmentación vertical.

En DYMOND se eliminan dichos problemas, ya que sólo se creará un nuevo EFV cuando los patrones de acceso a la base de datos hayan sufrido cambios mayores a los cambios promedio y cuando el costo del EFV actualmente implementado sea mayor a un umbral previamente determinado por DYMOND. Además, DYMOND si toma en cuenta el cambio en los atributos.

DYMOND utiliza el algoritmo MAVP y tiene un *Recolector de Estadísticas* que reúne toda la información que MAVP necesita para obtener un esquema de fragmentación vertical (EFV) eliminando el costo de la fase de análisis que debe realizar el administrador de la base de datos. Cuando el *Recolector de Estadísticas* detecta cambios importantes en los patrones de acceso y en el tamaño de los atributos dispara el *Evaluador de Desempeño*. Para detectar cuando debe iniciar el proceso de fragmentación DYMOND utiliza el modelo de costo propuesto para obtener un umbral de desempeño de la MMDB distribuida. Si el desempeño de la MMDB distribuida cae bajo dicho umbral, entonces el *Evaluador de*

Desempeño dispara a MAVP, el cual obtiene un nuevo EFV con base en la información almacenada por el *Recolector de Estadísticas*, y dispara un *Generador de Fragmentos* que materializa el nuevo EFV en la base de datos si el costo del nuevo EFV es menor que el costo del EFV actualmente implementado.

DYMOND se implementó en el DBMS PostgreSQL utilizando disparadores o triggers. Los disparadores le permiten a DYMOND:

- Iniciar el proceso de recolección de estadísticas, para esto se utiliza un disparador temporal, el cual puede ejecutarse automáticamente cada día, cada hora o cada minuto dependiendo de las necesidades del DBA.
- Recolectar la información necesaria en el proceso de fragmentación vertical, esta información incluye el nombre, tamaño y número de atributos, así como el número de consultas, su descripción, los atributos que usa y su frecuencia de ejecución.
- Detectar los cambios en las consultas y en los atributos, registrar dichos cambios y en caso de que sean mayores que los cambios promedio previamente almacenados disparar el *Evaluador de Desempeño*.
- Invocar al algoritmo MAVP si el costo obtenido por el *Evaluador de Desempeño* es mayor de un umbral previamente determinado por DYMOND.
- Actualizar las estadísticas de los atributos y de las consultas, así como el valor del umbral de desempeño cada vez que se realiza una nueva fragmentación vertical.

DYMOND se evaluó en PostgreSQL, y se demostró que genera un nuevo EFV que reduce el costo y el tiempo de ejecución de las consultas sin la intervención del DBA.

Otra contribución de este trabajo de tesis fue el desarrollo de un método de fragmentación vertical basado en soporte (SVP), el cual es una medida utilizada en minería de reglas de asociación. La principal ventaja de este método es que obtiene automáticamente el valor adecuado del umbral de soporte mínimo *min-sup* para generar el EFV óptimo. Esto es importante porque los algoritmos de fragmentación vertical basados en reglas de

asociación existentes no dan una pauta para fijar el valor del umbral, así que es necesario probar con varios valores hasta encontrar el umbral con el que se obtiene la mejor solución.

Los métodos de fragmentación existentes se clasificaron en estáticos y dinámicos en bases de datos tradicionales y en MMDBs. Se propusieron cuatro métodos de fragmentación, uno de cada tipo:

- SVP es un método de fragmentación vertical estático para bases de datos tradicionales, ya que es necesario que el DBA realice una etapa de análisis de requerimientos para obtener toda la información (frecuencia de las consultas y atributos utilizados) que este método requiere como entrada para generar el EFV. Posteriormente, el DBA debe materializar el EFV en la base de datos relacional.

- MAVP es una técnica de fragmentación vertical estática para bases de datos multimedia. El DBA obtiene la información de entrada para que MAVP determine el EFV. Sin embargo, esta información incluye también el tamaño de los atributos. También el DBA debe materializar el EFV en la base de datos multimedia.

- DYVEP es una propuesta de fragmentación vertical dinámica para bases de datos relacionales. DYVEP obtiene toda la información necesaria en el proceso de fragmentación vertical, genera el EFV y lo materializa por sí mismo. Si ocurren cambios en la base de datos relacional, entonces realiza una nueva fragmentación.

- DYMOND es un método de fragmentación vertical dinámica para bases de datos multimedia. Este método monitoriza la base de datos multimedia, si ocurren suficientes cambios, entonces dispara un evaluador de desempeño que obtiene el nuevo costo del EFV, cuando el costo es mayor que un umbral previamente determinado por DYMOND, se dispara MAVP para obtener el EFV. Finalmente, materializa el EFV.

9.2. Trabajo futuro

En el futuro se piensa realizar las siguientes actividades:

- Desarrollar un algoritmo de fragmentación híbrida para MMDBs, ya que la mayoría de las consultas en estas bases de datos sólo requieren un subconjunto tanto de atributos como de registros, por lo tanto si se reduce la cantidad de atributos y de registros irrelevantes accedidos por las consultas, se obtendrán esquemas de fragmentación que reducirán en gran manera el tiempo y costo de ejecución de las consultas.

- Tomar en cuenta consultas basadas en contenido (rango y k-vecinos más cercanos). La recuperación basada en contenido consiste en obtener información de la MMDB de acuerdo a características de los objetos multimedia como forma, textura y color. Esta normalemente se basa en la similitud en vez de una igualación exacta entre una consulta y un conjunto de elementos de la base de datos. Para responder estas consultas es necesario evaluar la similitud de cada objeto multimedia almacenado en la base de datos y el objeto a consultar, y recuperar los objetos más parecidos a la consulta. La fragmentación en estas bases de datos puede minimizar el número de objetos comparados y, por lo tanto, el tiempo de respuesta de las consultas.

- Probar DYMOND en el DBMS multimedia Oracle, en el DBMS orientados a objetos DB4O (DataBase for Objects) y en algún DBMS orientado a columnas [117].

Publicaciones

Durante el trabajo de tesis se publicaron los siguientes artículos:

1. Lisbeth Rodríguez and Xiaoou Li, A Vertical Partitioning Algorithm for Distributed Multimedia Databases, A. Hameurlain et al. (Eds.): DEXA 2011: 22nd International Conference on Database and Expert Systems Applications, Part II, LNCS 6861, pp. 544-558, Springer-Verlag, 2011.

2. Lisbeth Rodríguez and Xiaoou Li, A Dynamic Vertical Partitioning Approach for Distributed Database System, SMC 2011: IEEE International Conference on Systems, Man and Cybernetics, Anchorage, Alaska, pp. 1853-1858, 2011.

3. Lisbeth Rodríguez and Xiaoou Li, A Support-Based Vertical Partitioning Method for Database Design, CCE 2011: 8th International Conference on Electrical Engineering, Computing Science and Automatic Control.

4. Lisbeth Rodríguez, Xiaoou Li and Pedro Mejía-Álvarez, An Active System for Dynamic Vertical Partitioning of Relational Databases, I. Batyrshin and G. Sidorov (Eds.): MICAI 2011, Part II, LNAI 7095, pp. 273-284, Springer-Verlag, 2011.

5. Lisbeth Rodríguez and Xiaoou Li, Dynamic Vertical Partitioning of Multimedia Databases Using Active Rules, S. W. Liddle et al. (Eds.): DEXA 2012: 23rd International Conference on Database and Expert Systems Applications, Part II, LNCS 7447, pp. 191-198, Springer-Verlag, 2012.

6. Lisbeth Rodríguez and Xiaoou Li, DYMOND: An Active System for Dynamic Vertical Partitioning of Multimedia Databases, Bipin C. Desai, Jaroslav Pokorny, Jorge Bernardino (Eds.): IDEAS 2012: 16th International Database Engineering & Applications Symposium, August 8-10, Prague, Czech Republic.

Bibliografía

[1] Hector García Molina, Jeff Ullman, J.W.: Database Systems: The Complete Book. Prentice Hall (2008)

[2] Abraham Silverschatz, Henry F. Korth, S.S.: Database System Concepts. sixth edn. Mc Graw Hill (2011)

[3] de Vries, A.P.: Content and Multimedia Database Management Systems. PhD thesis, University of Twente (1999)

[4] Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13** (1970) 377–387

[5] Chamberlin, D.D., Boyce, R.F.: Sequel: A structured english query language. In: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control. SIGFIDET '74, New York, NY, USA, ACM (1974) 249–264

[6] Subramanya, S.R.: Multimedia databases. IEEE Potentials **18** (1999) 16–18

[7] Gemmell, D.J., Vin, H.M., Kandlur, D.D., Venkat Rangan, P., Rowe, L.A.: Multimedia storage servers: a tutorial. Computer **28** (1995) 40–49

[8] Ghafoor, A.: Multimedia database management: Perspectives and challenges. In: Proceedings of the 13th British National Conference on Databases: Advances in Databases. BNCOD 13, London, UK, Springer-Verlag (1995) 12–23

[9] Yu, C., Brandenburg, T.: Multimedia database applications: Issues and concerns for classrom teaching. The International Journal of Multimedia and Its Applications (IJMA) **3** (2011) 1–9

[10] Adjeroh, D.A., Nwosu, K.C.: Multimedia database management-requirements and issues. IEEE Multimedia **4** (1997) 24–33

[11] Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Third edn. Springer (2011)

[12] Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32** (2000) 422–469

[13] Ma, H.: Distribution Design for Complex Value Databases. PhD thesis, Massey University (2007)

[14] Hauglid, J.O., Ryeng, N.H., Nørvåg, K.: Dyfram: dynamic fragmentation and replica management in distributed database systems. Distrib. Parallel Databases **28** (2010) 157–185

[15] Fung, C.W., Leung, E.W.C., Li, Q.: Efficient query execution techniques in a 4dis video database system for elearning. Multimedia Tools Appl. **20** (2003) 25–49

[16] Saad, S., Tekli, J., Chbeir, R., Yetongnon, K.: Towards multimedia fragmentation. Lecture Notes in Computer Science **4152/2006** (2006) 415–429

[17] Getahun, F., Tekli, J., Atnafu, S., Chbeir, R.: Towards efficient horizontal multimedia database fragmentation using semantic-based predicates implication. XXII Simpósio Brasileiro de Banco de Dados (SBBD) (2007) 68–82

[18] Getahun, F., Tekli, J., Atnafu, S., Chbeir, R.: The use of semantic-based predicates implication to improve horizontal multimedia database fragmentation. In: MS '07: Workshop on multimedia information retrieval on The many faces of multimedia semantics, New York, NY, USA, ACM (2007) 29–38

[19] Chbeir, R., Laurent, D.: Towards a novel approach to multimedia data mixed fragmentation. In: MEDES '09: Proceedings of the International Conference on Management of Emergent Digital EcoSystems, New York, NY, USA, ACM (2009) 200–204

[20] Fung, C.W., Karlapalem, K., Li, Q.: An evaluation of vertical class partitioning for query processing in object-oriented databases. IEEE Transactions on Knowledge and Data Engineering **14** (2002) 1095–1118

[21] Fung, C.W., Karlapalem, K., Li, Q.: Cost-driven vertical class partitioning for methods in object oriented databases. The VLDB Journal **12** (2003) 187–210

[22] Lu, G.: Multimedia Database Management Systems. Artech House Computing Library (1999)

[23] van Doorn, M.G.L.M., de Vries, A.P.: The psychology of multimedia databases. In: Proceedings of the fifth ACM conference on Digital libraries. DL '00, New York, NY, USA, ACM (2000) 1–9

[24] Guinepain, S., Gruenwald, L.: Research issues in automatic database clustering. SIGMOD Rec. **34** (2005) 33–38

[25] Chaudhuri, S., Konig, A.C., Narasayya, V.: Sqlcm: a continuous monitoring framework for relational database engines. In: Proc. 20th Int Data Engineering Conf. (2004) 473–484

[26] Ramez Elmasri, S.B.N.: Fundamentos de Sistemas de Bases de Datos. Quinta edn. Pearson Addison Wesley (2007)

[27] Valduriez, P.: Principles of distributed data management in 2020? In: Proceedings of the 22nd international conference on Database and expert systems applications - Volume Part I. DEXA'11, Berlin, Heidelberg, Springer-Verlag (2011) 1–11

[28] Coronel Carlos, Steven Morris, P.R.: Bases de Datos Diseño, implementación y administración. Novena edn. Gengage Learning (2011)

[29] Gray, J.: Transparency in its place–the case against transparent access to geographically distributed data. In Stonebraker, M., ed.: Readings in database systems (2nd ed.). Morgan Kaufmann Publishers Inc. (1994) 592–602

[30] Levin, K.D., Morgan, H.L.: Optimizing distributed data bases: a framework for researc. In: Proceedings of the May 19-22, 1975, national computer conference and exposition. AFIPS '75, New York, NY, USA, ACM (1975) 473–478

[31] Ceri S., Pernici B., W.G.: Distributed database methodologies. In: Proceedings of the IEEE. Volume 75. (1987) 533–546

[32] Yao, S.B., Navathe, S.B., Weldon, J.L.: An integrated approach to database design. In: Proceedings of the NYU Symposium on Data Base Design Techniques I: Requirements and Logical Structures, London, UK, UK, Springer-Verlag (1982) 1–30

[33] S., C., G., P.: Distributed Databases Principles and System. McGraw-Hill, New York (1984)

[34] Grant, J.: Constraint preserving and lossless database transformations. Inf. Syst. **9** (1984) 2139–146

[35] Blankinship, R., Hevner, A.R., Yao, S.B.: An iterative method for distributed database design. In: Proceedings of the 17th International Conference on Very Large Data Bases. VLDB '91, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1991) 389–400

[36] Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. ACM Trans. Database Syst. **9** (1984) 680–710

[37] Navathe, S.B., Ra, M.: Vertical partitioning for database design: a graphical algorithm. In: Proceedings of the 1989 ACM SIGMOD international conference on Management of data. SIGMOD '89, New York, NY, USA, ACM (1989) 440–450

[38] Ibaraki, T., Kameda, T.: On the optimal nesting order for computing n-relational joins. ACM Trans. Database Syst. **9** (1984) 482–502

[39] S., S.M., B., Y.S.: Query optimization in distributed data base systems. Advances in Computers **21** (1982) 225–273

[40] Epstein, R., Stonebraker, M., Wong, E.: Distributed query processing in a relational data base system. In: Proceedings of the 1978 ACM SIGMOD international conference on management of data. SIGMOD '78, New York, NY, USA, ACM (1978) 169–180

[41] Page, Jr., T.W., Popek, G.J.: Distributed management in local area networks. In: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems. PODS '85, New York, NY, USA, ACM (1985) 135–142

[42] Ramakrishnan, R., Gehrke, J.: Database Management Systems. Second edn. Mc Graw Hill (1999)

[43] Seppälä, Y.: Definition of extraction files and their optimization by zero-one programming. BIT Numerical Mathematics **7** (1967) 206–215

[44] Babad, J.M.: A record and file partitioning model. Commun. ACM **20** (1977) 22–31

[45] Hammer, M., Niamir, B.: A heuristic approach to attribute partitioning. In: SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (1979) 93–101

[46] Niamir, B.: Attribute partitioning in a self-adaptive relational data base system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)

[47] Zhenjie, L.: Adaptive reorganization of database structures through dynamic vertical partitioning of relational tables. Master's thesis, University of Wollongong (2007)

[48] Weikum, G., Hasse, C., Mönkeberg, A., Zabback, P.: The comfort automatic tuning project. Inf. Syst. **19** (1994) 381–432

[49] Ray, C.: Distributed Database Systems. Pearson Education India (2009)

[50] Ceri, S., Negri, M., Pelagatti, G.: Horizontal data partitioning in database design. In: Proceedings of the 1982 ACM SIGMOD international conference on Management of data. SIGMOD '82, New York, NY, USA, ACM (1982) 128–136

[51] Shin, D.G., Irani, K.B.: Fragmenting relations horizontally using a knowledge-based approach. IEEE Trans. Softw. Eng. **17** (1991) 872–883

[52] Khalil, N., Eid, D., Khair, M.: Availability and reliability issues in distributed databases using optimal horizontal fragmentation. In Bench-Capon, T., Soda, G., Tjoa, A., eds.: Database and Expert Systems Applications. Volume 1677 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1999) 805–805

[53] Dimovski, A., Velinov, G., Sahpaski, D.: Horizontal partitioning by predicate abstraction and its application to data warehouse design. In: Proceedings of the 14th east European conference on Advances in databases and information systems. ADBIS'10, Berlin, Heidelberg, Springer-Verlag (2010) 164–175

[54] Karlapalem, K., Li, Q.: Partitioning schemes for object oriented databases. In: in Proceeding of the Fifth International Workshop on Research Issues in Data Engineering-Distributed Object Management, RIDE-DOM'95. (1995)

[55] Karlapalem, K., Li, Q., Vieweg, S.: Method induced partitioning schemes for object-oriented databases. In: ICDCS'96. (1996) 377–384

[56] Ezeife, C.I., Barker, K.: A comprehensive approach to horizontal class fragmentation in a distributed object based system. Distrib. Parallel Databases **3** (1995) 247–272

[57] Bellatreche, L., Karlapalem, K., Simonet, A.: Horizontal class partitioning in object-oriented databases. In Hameurlain, A., Tjoa, A., eds.: Database and Expert Systems Applications. Volume 1308 of Lecture Notes in Computer Science. Springer-Verlag (1997) 58–67

[58] Bellatreche, L., Karlapalem, K., Basak, G.K.: Horizontal class partitioning for queries in object-oriented databases. Technical Report HKUST-CS98-6, University of Science and Technology, Clear Water Bay Kowloon, Hong Kong (1998)

[59] Ezeife, C., Zheng, J.: Measuring the performance of database object horizontal fragmentation schemes. Database Engineering and Applications Symposium, International (1999) 408–414

[60] Baiao, F., Mattoso, M., Zaverucha, G.: Horizontal fragmentation in object dbms: New issues and performance evaluation. In: In Proceedings of the19 th IEEE International Performance, Computing and Communications Conference" (IPCCC 2000), IEEE CS, Press (2000) 108–114

[61] Navathe, S.B., Karlapalem, K., Ra, M.: A mixed fragmentation methodology for initial distributed database design. Journal of Computer and Software Engineering **3** (1995) 395–426

[62] Cheng, C.H., Lee, W.K., Wong, K.F.: A genetic algorithm-based clustering approach for database partitioning. IEEE Transactions on Systems, Man and Cybernetics **32** (2002) 215–230

[63] Darabant, A.S., Campan, A.: Semi-supervised learning techniques: k-means clustering in oodb fragmentation. In: Proc. Second IEEE Int. Conf. Computational Cybernetics ICCC 2004. (2004) 333–338

[64] Darabant, A.S., Campan, A., Moldovan, G., Grebla, H.: Ai clustering techniques: A new approach in horizontal fragmentation of classes with complex attributes and methods in object oriented databases. In: Proceedings of the International Conference on Theory and Applications of Mathematics and Informatics, Thessaloniki, Greece (2004)

[65] Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. SIGMOD '04, New York, NY, USA, ACM (2004) 359–370

[66] Sacca, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Trans. Database Syst. **10** (1985) 29–56

[67] Papadomanolakis, S., Ailamaki, A.: Autopart: automating schema design for large scientific databases using data partitioning. In: Proc. 16th Int Scientific and Statistical Database Management Conf. (2004) 383–392

[68] Hoffer, J.A., Severance, D.G.: The use of cluster analysis in physical data base design. In: Proceedings of the 1st International Conference on Very Large Data Bases. VLDB '75, New York, NY, USA, ACM (1975) 69–86

[69] Ceri, S., Pernici, B., Wiederhold, G.: Optimization problems and solution methods in the design of data distribution. Information Systems **14** (1989) 261 – 272

[70] Lin, X., Orlowska, M., Zhang, Y.: A graph based cluster approach for vertical partitioning in database design. Data & Knowledge Engineering **11** (1993) 151 – 169

[71] Marir, F., Najjar, Y., AlFaress, M., Abdalla, H.: An enhanced grouping algorithm for vertical partitioning problem in ddbs. In: Computer and information sciences, 2007. iscis 2007. 22nd international symposium on. (2007) 1 –6

[72] Abuelyaman, E.S.: An optimized scheme for vertical partitioning of a distributed database. IJCSNS International Journal of Computer Science and Network Security **8** (2008)

[73] Chakravarthy, S., Muthuraj, J., Varadarajan, R., Navathe, S.B.: An objective function for vertically partitioning relations in distributed databases and its analysis. Distrib. Parallel Databases **2** (1994) 183–207

[74] Son, J.H., Kim, M.H.: An adaptable vertical partitioning method in distributed systems. J. Syst. Softw. **73** (2004) 551–561

[75] Cornell, D.W., Yu, P.S.: An effective approach to vertical partitioning for physical design of relational databases. IEEE Trans. Softw. Eng. **16** (1990) 248–258

[76] Chu, W.W., Ieong, I.T.: A transaction-based approach to vertical partitioning for relational database systems. IEEE Trans. Softw. Eng. **19** (1993) 804–812

[77] Gorla, N., Betty, P.W.Y.: Vertical fragmentation in databases using data-mining technique. International Journal of Data Warehousing and Mining (IJDWM) **4** (2008) 35–53

[78] Rodríguez, L., Li, X.: A support-based vertical partitioning method for database design. In: 8th International Conference on Electrical Engineering Computing Science and Automatic Control (CCE). (2011)

[79] Du, J., Barker, K., Alhajj, R.: Attraction - a global affinity measure for database vertical partitioning. In: Proceedings of the IADIS International Conference WWW/Internet 2003, ICWI 2003, Algarve, Portugal (2003) 538–548

[80] Chen, M.S., Han, J., Yu, P.S.: Data mining: An overview from a database perspective. IEEE Trans. on Knowl. and Data Eng. **8** (1996) 866–883

[81] ling Shyu, M., ching Chen, S., Kashyap, R.L.: Generalized affinity-based association rule mining for multimedia database queries. Knowledge and Information Systems (KAIS): An International Journal **3** (2001) 319–337

[82] Tzanis, G., Berberidis, C.: Mining for mutually exclusive items in transaction databases. International Journal of Data Warehousing and Mining (IJDWM) **3** (2007) 45–59

[83] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1994) 487–499

[84] Alsuwaiyel, M.H.: Algorithms: design techniques and analysis. Volume 7 of Lecture Notes Series on Computing. World Scientific (1999)

[85] Ezeife, C.I., Barker, K.: Vertical class fragmentation in a distributed object based system. Technical report, University of Manitoba, Winnipeg, Manitoba, Canada (1994)

[86] Ezeife, C.I., Barker, K.: Vertical fragmentation for advanced object models in a distributed object based system. In: In Proc. 8th Int. Conf. on Computing and Information. IEEE. (1995)

[87] Ezeife, C.I., Barker, K.: Distributed object based design: Vertical fragmentation of classes. Distrib. Parallel Databases **6** (1998) 317–350

[88] Bellatreche, L., Simonet, A., Simonet, M.: Vertical fragmentation in distributed object database systems with complex attributes and methods. In: Proc. Workshop Seventh Int Database and Expert Systems Applications. (1996) 15–21

[89] Malinowski, E., Chakravarthy, S.: Fragmentation techniques for distributing object-oriented databases. In: Proceedings of the 16th International Conference on Conceptual Modeling. ER '97, London, UK, Springer-Verlag (1997) 347–360

[90] Fung, C.W., Karlapalem, K., Li, Q.: An analytical approach towards evaluating method-induced vertical partitioning algorithms. Technical Report HKUST-CS96-33, University of Science and Technology, Clear Water Bay, Kowloon, HONG KONG (1996)

[91] Cruz, F., Baião, F.A., Mattoso, M., Zaverucha, G.: Towards a theory revision approach for the vertical fragmentation of object oriented databases. In: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence: Advances in Artificial Intelligence. SBIA '02, London, UK, UK, Springer-Verlag (2002) 216–226

[92] Rajan, J., Saravanan, V.: Vertical partitioning in object oriented databases using intelligent agents. IJCSNS International Journal of Computer Science and Network Security **8** (2008) 205–210

[93] Fung, C.W., Karlapalem, K., Li, Q.: Cost-driven evaluation of vertical class partitioning in object-oriented databases. In: Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA), World Scientific Press (1997) 11–20

[94] Baiao, F., Mattoso, M.: A mixed fragmentation algorithm for distributed object oriented databases. In: ICCI'98: In Proceedings of the Int. Conf. on Computing and Information, Winnipeg, Canada (1998) 141–148

[95] Baiao, F.A., Mattoso, M., Zaverucha, G.: Towards an inductive design of distributed object oriented databases. In: Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems. COOPIS '98, Washington, DC, USA, IEEE Computer Society (1998) 188–197

[96] Baiao, F., Mattoso, M., Zaverucha, G.: A distribution design methodology for object dbms. Distrib. Parallel Databases **16** (2004) 45–90

[97] Ng, V., Law, D.M., Gorla, N., Chan, C.K.: Applying genetic algorithms in database partitioning. In: Proceedings of the 2003 ACM symposium on Applied computing. SAC '03, New York, NY, USA, ACM (2003) 544–549

[98] Kwok, Y.K., Karlapalem, K., Ahmad, I., Pun, N.M.: Design and evaluation of data allocation algorithms for distributed multimedia database systems. IEEE Journal on Selected Areas and Communications **14** (1996) 1332–1348

[99] McIver, Jr., W.J., King, R.: Self-adaptive, on-line reclustering of complex object data. In: Proceedings of the 1994 ACM SIGMOD international conference on Management of data. SIGMOD '94, New York, NY, USA, ACM (1994) 407–418

[100] Liroz-Gistau, M., Akbarinia, R., Pacitti, E., Porto, F., Valduriez, P.: Dynamic workload-based partitioning for large-scale databases. In Liddle, S., Schewe, K.D., Tjoa, A., Zhou, X., eds.: Database and Expert Systems Applications. Volume 7447 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 183–190

[101] Guinepain, S., Gruenwald, L.: Automatic database clustering using data mining. In: Proceedings of the 17th International Conference on Database and Expert Systems Applications, IEEE Computer Society (2006) 124–128

[102] Li, L., Gruenwald, L.: Autonomous database partitioning using data mining on single computers and cluster computers. In: Proceedings of the 16th International Database Engineering & Applications Sysmposium. IDEAS '12, New York, NY, USA, ACM (2012) 32–41

[103] Khan, S.I., Hoque, A.S.M.L.: A new technique for database fragmentation in distributed systems. International Journal of Computer Applications **5** (2010) 20–24 Published By Foundation of Computer Science.

[104] Son, J.H., Kim, M.H.: alpha-partitioning algorithm: Vertical partitioning based on the fuzzy graph. In: Proceedings of the 12th International Conference on Database and Expert Systems Applications. DEXA '01, London, UK, Springer-Verlag (2001) 537–546

[105] Sleit, A., AlMobaideen, W., Al-Areqi, S., Yahya, A.: A dynamic object fragmentation and replication algorithm in distributed database systems. American Journal of Applied Sciences **4** (2007) 613–618

[106] Chavarría-Báez, L., Li, X.: Structural error verification in active rule-based systems using petri nets. In: Proc. Fifth Mexican Int. Conf. Artificial Intelligence MICAI '06. (2006) 12–21

[107] Chavarría-Báez, L., Li, X.: Ecapnver: A software tool to verify active rule bases. In: Proc. 22nd IEEE Int Tools with Artificial Intelligence (ICTAI) Conf. Volume 2. (2010) 138–141

[108] Gay, J.Y., Gruenwald, L.: A clustering technique for object oriented databases. In: Proceedings of the 8th International Conference on Database and Expert Systems Applications. DEXA '97, London, UK, Springer-Verlag (1997) 81–90

[109] Darmont, J., Fromantin, C., Régnier, S., Gruenwald, L., Schneider, M.: Dynamic clustering in object-oriented databases: An advocacy for simplicity. In: International Workshop on Distributed Object Management/Workshop on Object-Oriented Database Systems. (2000) 71–85

[110] Berstel, B., Bonnard, P., Bry, F., Eckert, M., Pătrânjan, P.L.: Reactive rules on the web. In Antoniou, G.e.a., ed.: Reasoning Web. Volume 4636 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 183–239

[111] Chavarría-Báez, L., Li, X.: Termination analysis of active rules — a petri net based approach. In: Proc. IEEE Int. Conf. Systems, Man and Cybernetics SMC 2009. (2009) 2205–2210

[112] Jeniffer Widom, S.C., ed.: Active database systems Triggers and rules for advanced database processing. Morgan Kauffman Publishers, Inc. (1996)

[113] Li, X., Medina, J.M., Chapa, S.V.: Applying petri nets in active database systems. IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews **37** (2007) 482–493

[114] Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. **31** (1999) 63–103

[115] Transaction Processing Performance Council (TPC) San Francisco, CA: TPC BENCHMARK H, http://www.tpc.org/tpch/. (2008)

[116] Rodríguez, L., Li, X.: A vertical partitioning algorithm for distributed multimedia databases. In A Hameurlain, e.a., ed.: Proceedings of DEXA 2011. Volume 6861 of DEXA 2011., Springer Verlag (2011) 544–558

[117] Gu, X., Yang, X., Wang, W., Jin, Y., Meng, D.: Chac: An effective attribute clustering algorithm for large-scale data processing. Networking, Architecture, and Storage, International Conference on (2012) 94–98