# DISEÑO DE BASE DE DATOS DISTRIBUIDA

(Texto Base)

**Materia: Sistemas Distribuidos**

**AQUINO BOLIVIA – 2005**

**Cochabamba - Bolivia**

# CAPITULO 1. INTRODUCCION

La cantidad de innovaciones tecnológicas que ha habido en los últimos años ha promovido un cambio en la forma de observar a los sistemas de información y, en general, a las aplicaciones computacionales. Existen avances tecnológicos que se realizan continuamente en circuitos, dispositivos de almacenamiento, programas y metodologías. Sin embargo, los cambios tecnológicos van de la mano con la demanda de los usuarios y programas para la explotación exhaustiva de tales dispositivos mejorados. Por tanto, existe un continuo desarrollo de nuevos productos los cuales incorporan ideas nuevas desarrolladas por compañías e instituciones académicas.

Aún cuando es posible que un usuario común no perciba los desarrollos relevantes de nuevos productos, para las aplicaciones existe una demanda permanente por mayor funcionalidad, mayor número de servicios, más flexibilidad y mejor rendimiento. Así, al diseñar un nuevo sistema de información o al prolongar la vida de uno ya existente, se debe buscar siempre formas para enlazar las soluciones ofrecidas por la tecnología disponible a las necesidades de las aplicaciones de los usuarios.

Una área en la cual las soluciones están integrando tecnología con nuevas arquitecturas o formas de hacer las cosas es, sin lugar a dudas, el área de los sistemas distribuidos de información. Ellos se refieren al manejo de datos almacenados en facilidades de cómputo localizadas en muchos sitios ligados a través de una red de comunicaciones. Un caso específico de estos sistemas distribuidos es lo que se conoce como bases de datos distribuidas, tópico a estudiar en estas notas.

## *1. MOTIVACION*

Existen dos fuerzas que han impulsado la evolución de los sistemas de bases de datos. Por un lado los usuarios como parte de organizaciones más complejas han demandado una serie de capacidades que se han ido incorporando en los sistemas de bases de datos (Figura 1.1). Un ejemplo de esto es la necesidad de integrar información proveniente de fuentes diversas. Por otro lado, la tecnología ha hecho posible que algunas facilidades inicialmente imaginadas solo en sueños se conviertan en realidad. Por ejemplo, las transacciones en línea que permite el sistema bancario actual no hubiera sido posible sin el desarrollo de los equipos de comunicación. Los sistemas de cómputo distribuido son ejemplos claros en donde presiones organizacionales se combinan con la disponibilidad de nuevas tecnologías para hacer realidad tales aplicaciones.

### 1.1 La presión por datos distribuidos

**La presión de los usuarios**

Las bases de datos grandes permiten organizar la información relevantes a alguna parte de la operación de una organización como por ejemplo servicios de salud, corporaciones industriales o bancos. Casi cualquier organización que ha incorporado sistemas de información para su funcionamiento ha experimentado dos fases.

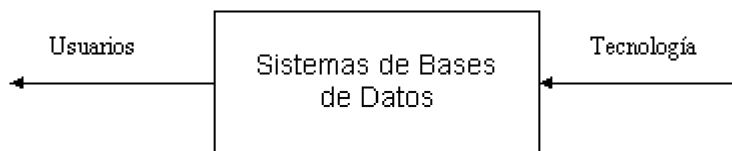



Figura 1.1. Fuerzas evolucionarias en los sistemas de bases de datos.

En la primera fase, se ha agrupando toda la información en un solo lugar. La idea original era que todos los accesos a datos podrían ser integrados en un solo lugar usando herramientas de bases de datos tales como lenguajes de descripción de datos, lenguajes de manipulación de datos, mecanismos de acceso, verificadores de restricciones y lenguajes de alto nivel. Para poder tener estos mecanismos de almacenamiento y recuperación de información, las organizaciones hicieron fuertes inversiones en equipos computacionales sofisticas y con grandes capacidades. Sin embargo, después de experimentar por un tiempo con este enfoque, muchas organizaciones encontraron que el sistema completo era satisfactorio, en algún grado, para un buen número de usuarios pero muy pocos obtenían un servicio óptimo. Más aún, bajo este esquema centralizado los "propietarios" u originadores de la información específica perdieron el control sobre el manejo de su información ya que ésta no se almacenaba en sus lugares de trabajo.

Algunos experimentos mostraron que el 90% de las operaciones de entrada y salida de información eran "locales" (correspondientes al departamento que las generaba) y solo el 10% de tales operaciones involucraba información cruzada (información proveniente de más de un departamento). Así, en la segunda fase se promovió la descentralización de los sistemas de bases de datos corporativos. Entonces, se empezaron a adquirir sistemas de software y hardware departamentales. Este enfoque presentó grandes beneficios para el control de la seguridad de la información y la disponibilidad de la misma. Permitió que los esquemas de mantenimiento y planeación de los sistemas de información afectara en menor medida al funcionamiento general de la organización.

Sin embargo, muy pronto empezaron a aparecer inconvenientes con este enfoque. Se presentaron problemas de consistencia de la información entre los sistemas locales y central y se hallaron dificultados al transferir información de entre departamentos diferentes de una corporación.

De esta manera, en una tercera fase (la cual aún no ha concluido) se ha tratado de formalizar la descentralización de las bases de datos y de sus funciones manteniendo la integridad de la información y quizá algún tipo de control centralizado o distribuido.

**La presión de la tecnología**

Existen buenas razones técnicas para distribuir datos. La más obvia es la referente a la sobrecarga de los canales de entrada y salida a los discos en donde se almacena finalmente la información. Es mucho mejor distribuir los accesos a la información sobre diferentes canales que concentrarlos en uno solo. Otra razón de peso es que las redes de computadoras empezaron a trabajar a velocidades razonables abriendo la puerta a la distribución del trabajo y la información.

El hacer una descentralización de la información se justifica desde el punto de vista tecnológico por las siguientes razones:

- Para permitir autonomía local y promover la evolución de los sistemas y los cambios en los requerimientos de usuario.
- Para proveer una arquitectura de sistemas simple, flexible y tolerante a fallas.
- Para ofrecer buenos rendimientos.

Existen aplicaciones que nacieron distribuidas. Para ellas ha sido necesario el uso de nuevas tecnologías para integrar sistemas de información diferentes, de forma que, no se afecte de manera sustancial el estilo de trabajo o de hacer las cosas de los usuarios.

Aunque la idea de distribución de datos es bastante atractiva, su realización conlleva la superación de una serie de dificultades tecnológicas entre las que se pueden mencionar:

- Asegurar que el acceso entre diferentes sitios o nodos y el procesamiento de datos se realice de manera eficiente, presumiblemente óptima.

- Transformar datos e integrar diferentes tipos de procesamiento entre nodos de un ambiente distribuido.

- Distribuir datos en los nodos del ambiente distribuido de una manera óptima.

- Controlar el acceso a los datos disponibles en el ambiente distribuido.

- Soportar la recuperación de errores de diferentes módulos del sistema de manera segura y eficiente.

- Asegurar que los sistemas locales y globales permanezcan como una imagen fiel del mundo real evitando la interferencia destructiva que pueden ocasionar diferentes transacciones en el sistema.

Así también, la aplicación de técnicas de distribución de información requiere de superar algunas dificultades de índole organizacional y algunas otras relacionadas con los usuarios. Entre ellas se puede mencionar:

- El desarrollo de modelos para estimar la capacidad y el tráfico esperado en el sistema distribuido.

- Soportar el diseño de sistemas de información distribuidos. Por ejemplo, ayudar a decidir donde localizar algún dato particular o donde es mejor ejecutar un programa de aplicación.

- Considerar la competencia que habrá por el uso de los recursos entre nodos diferentes.

Aun cuando las dificultades mencionadas son importantes, las ventajas de la distribución de información han promovido su aplicación en ambientes del presente y del futuro.

### 1.2 Heterogeneidad y la presión para integrar datos

La descentralización de los sistemas de información y el advenimiento de los sistemas distribuidos están bien justificados. Sin embargo, existe todavía un argumento importante para el desarrollo de sistemas de bases de datos distribuidas; éste se refiere a la integración de necesidades de procesamiento *no locales* en donde es necesario intercambiar información proveniente de otras áreas o departamentos. La descentralización de la información promueve la heterogeneidad en su manejo. La heterogeneidad se puede dar a muchos niveles, desde la forma y significado de cada dato hasta el formato y el medio de almacenamiento que se elige para guardarlo. La integración de la información es de importancia mayor para el funcionamiento de una organización.

En resumen, en los sistemas de bases de datos distribuidas se persigue la integración de sistemas de bases de datos diversos no necesariamente homogéneos para dar a los usuarios una visión global de la información disponible. Este proceso de integración no implica la centralización de la información, más bien, con la ayuda de la tecnología de redes de computadoras disponible, la información se mantiene distribuida (localizada en diversos lugares) y los sistemas de bases de datos distribuidos permiten el acceso a ella como si estuviera localizada en un solo lugar. La distribución de la información permite, entre otras cosas, tener accesos rápidos a la información, tener copias de la información para accesos más rápidos y para tener respaldo en caso de fallas.

## 1.3 Computación Distribuida

Los sistemas de bases de datos distribuidas son un caso particular de los ***sistemas de cómputo distribuido*** en los cuales un conjunto de elementos de procesamiento autónomos (no necesariamente homogéneos) se interconectan por una red de comunicaciones y cooperan entre ellos para realizar sus tareas asignadas. Históricamente, el cómputo distribuido se ha estudiado desde muchos puntos de vista. Así, es común encontrar en la literatura un gran número de términos que se han usado para identificarlo. Entre los términos más comunes que se utilizan para referirse al cómputo distribuido podemos encontrar: funciones distribuidas, procesamiento distribuido de datos, multiprocesadores, multicomputadoras, procesamiento satelital, procesamiento tipo "backend", computadoras dedicadas y de propósito específico, sistemas de tiempo compartido, sistemas funcionalmente modulares.

Existen muchas componentes a distribuir para realizar una tarea. En computación distribuida los elementos que se pueden distribuir son:

- Control. Las actividades relacionadas con el manejo o administración del sistema.
- Datos. La información que maneja el sistema.
- Funciones. Las actividades que cada elemento del sistema realiza.
- Procesamiento lógico. Las tareas específicas involucradas en una actividad de procesamiento de información.

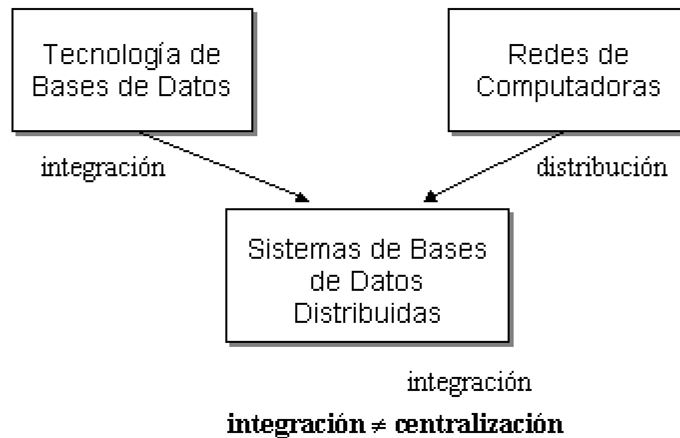



Figura 1.2. Motivación de los sistemas de bases de datos distribuidos.

## 1.4 Sistemas de bases de datos distribuidas

Una **base de datos distribuida** (BDD) es un conjunto de múltiples bases de datos *lógicamente relacionadas* las cuales se encuentran distribuidas entre diferentes sitios interconectados por una red de comunicaciones (ver Figura 1.2).

Un **sistema de bases de datos distribuida** (SBDD) es un sistema en el cual múltiples sitios de bases de datos están ligados por un sistema de comunicaciones, de tal forma que, un usuario en cualquier sitio puede accesar los datos en cualquier parte de la red exactamente como si los datos estuvieran almacenados en su sitio propio.

Un **sistema de manejo de bases de datos distribuidas** (SMBDD) es aquel que se encarga del manejo de la BDD y proporciona un mecanismo de acceso que hace que la distribución sea *transparente* a los

usuarios. El término transparente significa que la aplicación trabajaría, desde un punto de vista lógico, como si un solo SMBD ejecutado en una sola máquina, administrara esos datos.

Un **sistema de base de datos distribuida** (SBDD) es entonces el resultado de la integración de una base de datos distribuida con un sistema para su manejo.

Dada la definición anterior, es claro que algunos sistemas no se pueden considerar como SBDD. Por ejemplo, un sistema de tiempo compartido no incluye necesariamente un sistema de manejo de bases de datos y, en caso de que lo haga, éste es controlado y administrado por una sola computadora.

Un sistema de multiprocesamiento puede administrar una base de datos pero lo hace usualmente a través de un solo sistema de manejo de base de datos; los procesadores se utilizan para distribuir la carga de trabajo del sistema completo o incluso del propio SMBD pero actuando sobre una sola base de datos. Finalmente, una base de datos la cual reside en un solo sitio de una red de computadoras y que es accesada por todos los nodos de la red no es una base de datos distribuida (Figura 1.3). Este caso se trata de una base de datos cuyo control y administración esta centralizada en un solo nodo pero se permite el acceso a ella a través de la red de computadoras.

El medio ambiente típico de un SMBDD consiste de un conjunto de sitios o nodos los cuales tiene un sistema de procesamiento de datos completo que incluye una base de datos local, un sistema de manejo de bases de datos y facilidades de comunicaciones. Si los diferentes sitios pueden estar geográficamente dispersos, entonces, ellos están interconectados por una red de tipo WAN. Por otro lado, si los sitios están localizados en diferentes edificios o departamentos de una misma organización pero geográficamente en la misma ubicación, entonces, están conectados por una red local (LAN) (Figura 1.4).

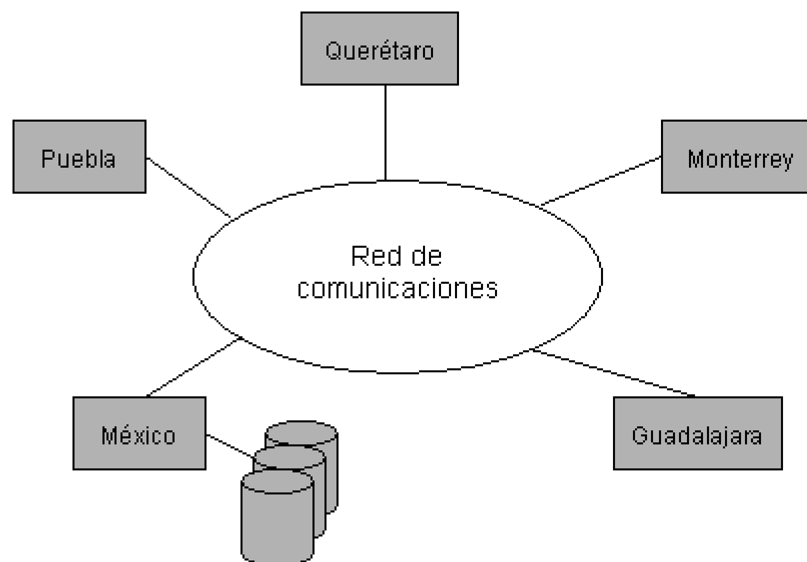



Figura 1.3. Un sistema centralizado sobre una red.

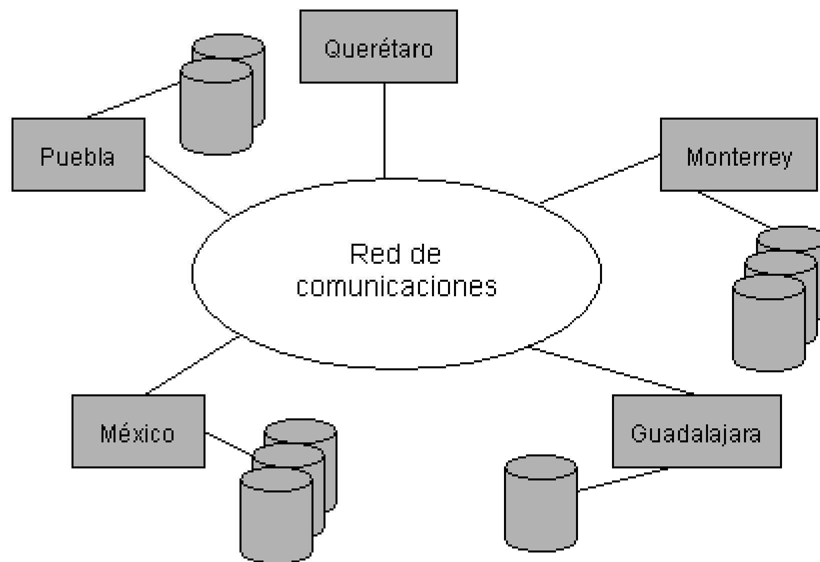



Figura 1.4. Un medio ambiente distribuido para bases de datos.

### 1.4.1 Ambientes con múltiples procesadores

Desde el punto de vista de las bases de datos, conceptualmente existen tres tipos de ambientes que se integran con múltiples procesadores:

1. Arquitecturas de memoria compartida. Consisten de diversos procesadores los cuales accesan una misma memoria y un misma unidad de almacenamiento (uno o varios discos). Algunos ejemplos de este tipo son las computadoras Sequent Encore y los mainframes IBM4090 y Bull DPS8 (Figura 1.5).

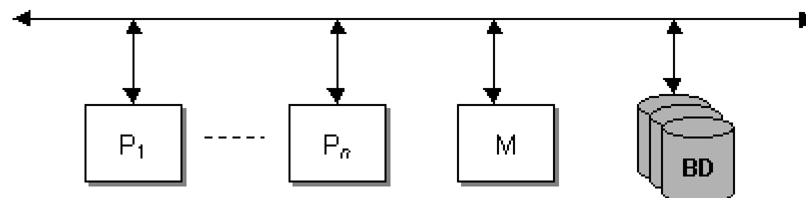



Figura 1.5. Arquitectura de memoria compartida.

2. Arquitecturas de disco compartido. Consiste de diversos procesadores cada uno de ellos con su memoria local pero compartiendo una misma unidad de almacenamiento (uno o varios discos). Ejemplos de estas arquitecturas son los cluster de Digital, y los modelos IMS/VS Data Sharing de IBM (Figura 1.6).

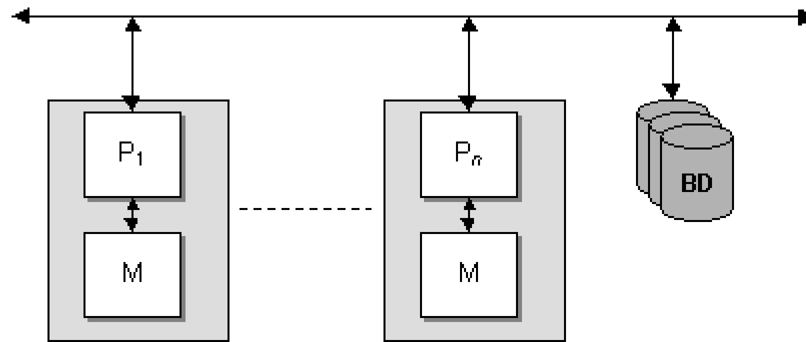



Figura 1.6. Arquitectura de disco compartido.

3. Arquitecturas nada compartido. Consiste de diversos procesadores cada uno con su propia memoria y su propia unidad de almacenamiento. Aquí se tienen los clusters de estaciones de trabajo, la computadoras Intel Paragon, NCR 3600 y 3700 e IBM SP2 (Figura 1.7).

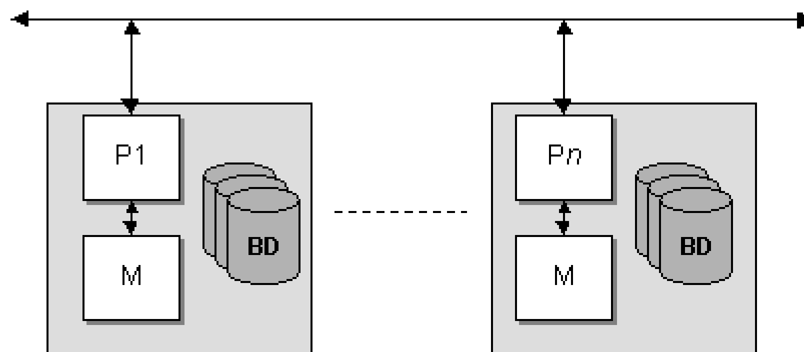



Figura 1.7. Arquitectura nada compartido.

### 1.4.2 Aplicaciones

Los ambientes en los que se encuentra con mayor frecuencia el uso de las bases de datos distribuidas son:

- Cualquier organización que tiene una estructura descentralizada.
- Casos típicos de lo anterior son: organismos gubernamentales y/o de servicio público.
- La industria de la manufactura, particularmente, aquella con plantas múltiples. Por ejemplo, la industria automotriz.
- Aplicaciones de control y comando militar.
- Líneas de transportación aérea.
- Cadenas hoteleras.
- Servicios bancarios y financieros.

### 1.4.3 Ventajas

Los SMBDD tienen múltiples ventajas. En primer lugar los datos son localizados en lugar más cercano, por tanto, el acceso es más rápido, el procesamiento es rápido debido a que varios nodos intervienen en el procesamiento de una carga de trabajo, nuevos nodos se pueden agregar fácil y rápidamente. La comunicación entre nodos se mejora, los costos de operación se reducen, son amigables al usuario, la probabilidad de que una falla en un solo nodo afecte al sistema es baja y existe una autonomía e independencia entre los nodos.

Las razones por las que compañías y negocios migran hacia bases de datos distribuidas incluyen razones organizacionales y económicas, para obtener una interconexión confiable y flexible con las bases de datos existente, y por un crecimiento futuro. El enfoque distribuido de las bases de datos se adapta más naturalmente a la estructura de las organizaciones. Además, la necesidad de desarrollar una aplicación global (que incluya a toda la organización), se resuelva fácilmente con bases de datos distribuidas. Si una organización crece por medio de la creación de unidades o departamentos nuevos, entonces, el enfoque de bases de datos distribuidas permite un crecimiento suave.

Los datos se pueden colocar físicamente en el lugar donde se accesan más frecuentemente, haciendo que los usuarios tengan control local de los datos con los que interactúan. Esto resulta en una autonomía local de datos permitiendo a los usuarios aplicar políticas locales respecto del tipo de accesos a sus datos.

Mediante la replicación de información, las bases de datos distribuidas pueden presentar cierto grado de tolerancia a fallas haciendo que el funcionamiento del sistema no dependa de un solo lugar como en el caso de las bases de datos centralizadas.

### 1.4.4 Desventajas

La principal desventaja se refiere al control y manejo de los datos. Dado que éstos residen en muchos nodos diferentes y se pueden consultar por nodos diversos de la red, la probabilidad de violaciones de seguridad es creciente si no se toman las precauciones debidas.

La habilidad para asegurar la integridad de la información en presencia de fallas no predecibles tanto de componentes de hardware como de software es compleja. La integridad se refiere a la consistencia, validez y exactitud de la información.

Dado que los datos pueden estar replicados, el control de concurrencia y los mecanismos de recuperación son mucho más complejos que en un sistema centralizado.

## 1.5 Aspectos importantes de los SMBD distribuidos

Existen varios factores relacionados a la construcción de bases de datos distribuidas que no se presentan en bases de datos centralizadas. Entre los más importantes se encuentran los siguientes:

1. **Diseño de la base de datos distribuida**. En el diseño de bases de datos distribuidas se debe considerar el problema de como distribuir la información entre diferentes sitios. Existen razones organizacionales las cuales determinan en gran medida lo anterior. Sin embargo, cuando se busca eficiencia en el acceso a la información, se deben abordar dos problemas relacionados. Primero, como fragmentar la información. Segundo, como asignar cada fragmento entre los diferentes sitios de la red. En el diseño de la BDD también es importante considerar si la información está replicada, es decir, si existen copias múltiples del mismo dato y, en este caso, como mantener la consistencia de la información. Finalmente, una parte importante en el diseño de una BDD se refiere al manejo del directorio. Si existen únicamente usuarios globales, se debe

manejar un solo directorio global. Sin embargo, si existen también usuarios locales, el directorio combina información local con información global.

2. **Procesamiento de consultas**. El procesamiento de consultas es de suma importancia en bases de datos centralizadas. Sin embargo, en BDD éste adquiere una relevancia mayor. El objetivo es convertir transacciones de usuario en instrucciones para manipulación de datos. No obstante, el orden en que se realizan las transacciones afecta grandemente la velocidad de respuesta del sistema. Así, el procesamiento de consultas presenta un problema de optimización en el cual se determina el orden en el cual se hace la menor cantidad de operaciones. Este problema de optimización es NP-difícil, por lo que en tiempos razonables solo se pueden obtener soluciones aproximadas. En BDD se tiene que considerar el procesamiento local de una consulta junto con el costo de transmisión de información al lugar en donde se solicitó la consulta.
3. **Control de concurrencia**. El control de concurrencia es la actividad de coordinar accesos concurrentes a la base de datos. El control de concurrencia permite a los usuarios accesar la base de datos en una forma multiprogramada mientras se preserva la ilusión de que cada usuario está utilizándola solo en un sistema dedicado. El control de concurrencia asegura que transacciones múltiples sometidas por usuarios diferentes no interfieran unas con otras de forma que se produzcan resultados incorrectos. En BDD el control de concurrencia es aún más complejo que en sistemas centralizados. Los algoritmos más utilizados son variaciones de aquellos usados en sistemas centralizados: candados de dos fases, ordenamiento por estampas de tiempo, ordenamiento por estampas de tiempo múltiples y control de concurrencia optimista. Un aspecto interesante del control de concurrencia es el manejo de interbloqueos. El sistema no debe permitir que dos o más transacciones se bloqueen entre ellas.
4. **Confiabilidad**. En cualquier sistema de bases de datos, centralizado o distribuido, se debe ofrecer garantías de que la información es confiable. Así cada consulta o actualización de la información se realiza mediante transacciones, las cuales tienen un inicio y fin. En sistemas distribuidos, el manejo de la atomicidad y durabilidad de las transacciones es aún más complejo, ya que una sola transacción puede involucrar dos o más sitios de la red. Así, el control de recuperación en sistemas distribuidos debe asegurar que el conjunto de agentes que participan en una transacción realicen todos un compromiso (commit) al unísono o todos al mismo tiempo restablezcan la información anterior (roll-back).

En la Figura 1.8 se presenta un diagrama con las relaciones entre los aspectos relevantes sobre las BDD.

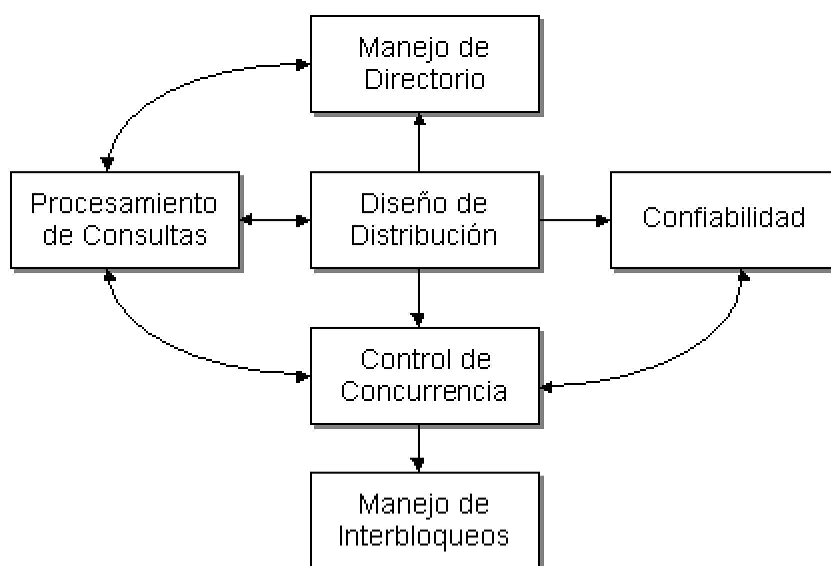



Figura 1.8. Factores importantes en BDD.

### 1.6 Estado del Arte

Aun cuando los beneficios del uso de BDD son claramente perceptibles, en la actualidad muchos de los desarrollos se encuentran únicamente en sistemas experimentales (de investigación). A continuación se discute el estado actual de las bases de datos comerciales respecto de cuatro logros potenciales asequibles en BDD.

1. **Manejo transparente de datos distribuidos, fragmentados y replicados**. Comercialmente aún no se soporta la replicación de información. La fragmentación utilizada es únicamente de tipo horizontal (ésta se discute en el capítulo 3). La distribución de información no se realiza aún con la transparencia requerida. Por ejemplo, el usuario debe indicar la localización de un objeto y el acceso a los datos es mediante sesiones remotas a bases de datos locales. La mayoría de los sistemas comerciales utilizan el modelo múltiples clientes-un solo servidor.
2. **Mejoramiento de la confiabilidad y disponibilidad de la información mediante transacciones distribuidas**. Algunos sistemas como Ingres, NonStop SQL y Oracle V 7.x ofrecen el soporte de transacciones distribuidas. En Sybase, por ejemplo, es posible tener transacciones distribuidas pero éstas deber ser implementadas en las aplicaciones mediante primitivas dadas. Respecto del soporte para replicación de información o no se ofrece o se hace a través de la regla une-lee-todos-escriben.
3. **Mejoramiento de la eficiencia**. Una mayor eficiencia es una de las grandes promesas de los SMBDD. Existen varias partes en donde ésto se puede lograr. En primer lugar, la ubicación de los datos a lugares próximos a donde se usan puede mejorar la eficiencia en el acceso a la información. Sin embargo, para lograrlo es necesario tener un buen soporte para fragmentación y replicación de información. Otro punto en donde se puede incrementar la eficiencia es mediante la explotación del paralelismo entre operaciones. Especialmente en el caso de varias consultas independientes, éstas se pueden procesar por sitios diferentes. Más aún, el procesamiento de una sola consulta puede involucrar varios sitios y así procesarse de manera más rápida. Sin embargo, la explotación del paralelismo requiere que se tenga tanta información requerida por cada aplicación en el sitio donde la aplicación se utiliza, lo cual conduciría a una replicación completa, esto es, tener toda la información en cada sitio de la red. El manejo de réplicas es complicado dado que las actualizaciones a este tipo de datos involucran a todos los sitios teniendo copias del dato. Los sistemas comerciales ofrecen únicamente aproximaciones a este requisito. Por ejemplo, en los bancos se destina usualmente el horario de oficina para hacer lecturas y las horas no hábiles para hacer actualizaciones. Otra estrategia es tener dos bases de datos, una para consultas y otra para actualizaciones.
4. **Mejor escalabilidad de las BD**. El tener sistemas escalables de manera fácil y económica se ha logrado por el desarrollo de la tecnología de microprocesadores y estaciones de trabajo. Sin embargo, respecto de la escalabilidad, la comunicación de la información tiene un costo el cual no se ha estudiado con suficiente profundidad.

# CAPITULO 2. ARQUITECTURA DE BASE DE DATOS DISTRIBUIDA

En el presente capítulo se mostrará la arquitectura general de un sistema de bases de datos distribuida, se introducirá el concepto de fragmentación de datos relacionado con el nivel de transparencia de distribución que un SBDD debe ofrecer. Se dará una descripción acerca de las componentes de las bases de datos distribuidas.

La arquitectura define la estructura de un sistema. Al definir la arquitectura se deben identificar las componentes de un sistema, las funciones que realiza cada una de las componentes y las interrelaciones e interacciones entre cada componente.

## 2.1 NIVELES DE TRANSPARENCIA EN SBDD

El próposito de establecer una arquitectura de un sistema de bases de datos distribuidas es ofrecer un nivel de transparencia adecuado para el manejo de la información. La ***transparencia*** se puede entender como la separación de la semántica de alto nivel de un sistema de las aspectos de bajo nivel relacionados a la implementación del mismo. Un nivel de transparencia adecuado permite ocultar los detalles de implementación a las capas de alto nivel de un sistema y a otros usuarios.

En sistemas de bases de datos distribuidos el propósito fundamental de la transparencia es proporcionar *independencia de datos* en el ambiente distribuido. Se pueden encontrar diferentes aspectos relacionados con la transparencia. Por ejemplo, puede existir transparencia en el manejo de la red de comunicación, transparencia en el manejo de copias repetidas o transparencia en la distribución o fragmentación de la información.

La ***independencia de datos*** es la inmunidad de las aplicaciones de usuario a los cambios en la definición y/u organización de los datos y viceversa. La independencia de datos se puede dar en dos aspectos: lógica y física.

1. ***Independencia lógica de datos***. Se refiere a la inmunidad de las aplicaciones de usuario a los cambios en la estructura lógica de la base de datos. Esto permite que un cambio en la definición de un esquema no debe afectar a las aplicaciones d eusuario. Por ejemplo, el agregar un nuevo atributo a una relación, la creación de una nueva relación, el reordenamiento lógico de algunos atributos.
2. ***Independencia física de datos***. Se refiere al ocultamiento de los detalles sobre las estructuras de almacenamiento a las aplicaciones de usuario. Esto es, la descripción física de datos puede cambiar sin afectar a las aplicaciones de usuario. Por ejemplo, los datos pueden ser movidos de un disco a otro, o la organización de los datos puede cambiar.

La ***transparencia al nivel de red*** se refiere a que los datos en un SBDD se accesan sobre una red de computadoras, sin embargo, las aplicaciones no deben notar su existencia. La transparencia al nivel de red conlleva a dos cosas:

1. ***Transparencia sobre la localización de datos***. Esto es, el comando que se usa es independiente de la ubicación de los datos en la red y del lugar en donde la operación se lleve a cabo. Por ejemplo, en Unix existen dos comandos para hacer una copia de archivo. Cp se utiliza para copias locales y rcp se utiliza para copias remotas. En este caso no existe transparencia sobre la localización.
2. ***Transparencia sobre el esquema de nombramiento***. Lo anterior se logra proporcionando un nombre único a cada objeto en el sistema distribuido. Así, no se debe mezclar la información de la localización con en el nombre de un objeto.

La ***transparencia sobre replicación*** de datos se refiere a que si existen réplicas de objetos de la base de datos, su existencia debe ser controlada por el sistema no por el usuario. Se debe tener en cuenta que al cuando el usuario se encarga de manejar las réplicas en un sistema, el trabajo de éste es mínimo por lo que se puede obtener una eficiencia mayor. Sin embargo, el usuario puede olvidarse de mantener la consistencia de las réplicas teniendo así datos diferentes.

La ***transparencia a nivel de fragmentación de datos*** permite que cuando los objetos de la bases de datos están fragmentados, el sistema tiene que manejar la conversión de consultas de usuario definidas sobre relaciones globales a consultas definidas sobre fragmentos. Así también, será necesario mezclar las respuestas a consultas fragmentadas para obtener una sola respuesta a una consulta global. El acceso a una base de datos distribuida debe hacerse en forma transparente.

**Ejemplo 2.1**. Como un ejemplo se utilizará a lo largo de estas notas una base de datos que modela una compañía de ingeniería. Las entidades a ser modeladas son ingenieros y proyectos. Para cada ingeniero, se desea conocer su número de empleado (ENO), su nombre (ENOMBRE), el puesto ocupado en compañía (TITULO), el salario (SAL), la identifiación de los nombres de proyectos en los cuales está trabajando (JNO), la responsabilidad que tiene dentro del proyecto (RESP) y la duración de su responsabilidad en meses (DUR). Similarmente, para cada proyecto se desea conocer el número de proyecto (JNO), el nombre del proyecto (JNOMBRE), el presupuesto asignado al proyecto (PRESUPUESTO) y el lugar en donde se desarrolla el proyecto (LUGAR).

Un ingeniero puede participar en más de un proyecto pero su salario corresponde únicamente al puesto que ocupa en la compañía. Así, después de aplicar normalización se obtienen las relaciones E –para ingenieros, J –para proyectos, S –para los salarios asignados a los puestos y G –para los ingenieros asignados a cada proyecto. Un ejemplo de las instancias para cada relación se presenta en la Figura 2.1.

E

| ENO | ENOMBRE | TITULO |
|---|---|---|
| E1 | Juan Rodríguez | Ingeniero Eléctrico |
| E2 | Miguel Sánchez | Analista de Sistemas |
| E3 | Armando Legarreta | Ingeniero Mecánico |
| E4 | Beatriz Molleda | Programador |
| E5 | Jorge Castañeda | Analista de Sistemas |
| E6 | Luis Chávez | Ingeniero Eléctrico |
| E7 | Roberto Dávila | Ingeniero Mecánico |
| E8 | Julia Jiménez | Analista de Sistemas |

G

| ENO | JNO | PUESTO | DUR |
|---|---|---|---|
| E1 | J1 | Administrador | 12 |
| E2 | J1 | Analista | 24 |
| E2 | J2 | Analista | 6 |
| E3 | J3 | Consultor | 10 |

| | | | |
|---|---|---|---|
| E3 | J4 | Ingeniero | 48 |
| E4 | J2 | Programador | 18 |
| E5 | J2 | Administrador | 24 |
| E6 | J4 | Administrador | 48 |
| E7 | J3 | Ingeniero | 36 |
| E7 | J5 | Ingeniero | 23 |
| E8 | J3 | Administrador | 40 |

J

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J1 | Instrumentación | 150000 | Monterrey |
| J2 | Desarrollo de bases de datos | 135000 | México |
| J3 | CAD/CAM | 250000 | Puebla |
| J4 | Mantenimiento | 310000 | México |
| J5 | CAD/CAM | 500000 | Monterrey |

S

| TITULO | SALARIO |
|---|---|
| Ingeniero Eléctrico | 40000 |
| Analista de Sistemas | 34000 |
| Ingeniero Mecánico | 27000 |
| Programador | 24000 |

Figura 2.1. Bases de datos de una empresa con cuatro relaciones.

Si se quisiera obtener todos los empleados y sus salarios en la corporación quienes han trabajado más de 12 meses se haría la consulta siguiente en SQL:

```
SELECT ENOMBRE, SALARIO

FROM E, G, S

WHERE JORNADA > 12 AND

E.ENO = G.ENO AND

E.TILE = S.TITLE
```

Se debe tener en cuenta que en cada sitio de la corporación puede haber esquemas diferentes o repetidos. Por ejemplo, en la Figura 2.2 se presentan esquemas diferentes para el manejo de proyectos, empleados y puestos en cada sitio de la bases de datos del Ejemplo 2.1.

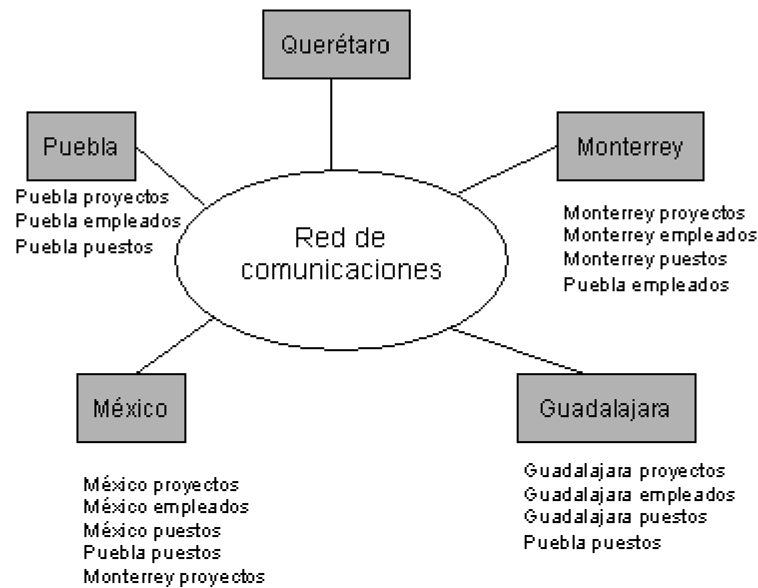



Figura 2.2. Diferentes sitios de un corporación.

En resumen, la transparencia tiene como punto central la independencia de datos. Los diferentes niveles de transparencia se puede organizar en capas como se muestra en la Figura 2.3. En el primer nivel se soporta la transparencia de red. En el segundo nivel se permite la transparencia de replicación de datos. En el tercer nivel se permite la transparencia de la fragmentación. Finalmente, en el último nivel se permite la transparencia de acceso (por medio de lenguaje de manipulación de datos).

La responsabilidad sobre el manejo de transparencia debe estar compartida tanto por el sistema operativo, el sistema de manejo de bases de datos y el lenguaje de acceso a la base de datos distribuida. Entre estos tres módulos se deben resolver los aspectos sobre el procesamiento distribuido de consultas y sobre el manejo de nombres de objetos distribuidos.

Figura 2.3. Organización en capas de los niveles de transparencia.

## 2.2 ARQUITECTURA DE UN SISTEMA DE BASES DE DATOS DISTRIBUIDAS

La mayoría de los sistemas de manejo de bases de datos disponibles actualmente están basadas en la arquitectura ANSI-SPARC la cual divide a un sistema en tres niveles: interno, conceptual y externo, como se puede apreciar en la Figura 2.4.

La vista conceptual, conocida también como vista lógica global, representa la visión de la comunidad de usuarios de los datos en la base de datos. No toma en cuenta la forma en que las aplicaciones individuales observan los datos o como éstos son almacenados. La vista conceptual está basada en el esquema conceptual y su construcción se hace en la primera fase del diseño de una base de datos.

Los usuarios, incluyendo a los programadores de aplicaciones, observan los datos a través de un esquema externo definido a nivel externo. La vista externa proporciona una ventana a la vista conceptual lo cual permite a los usuarios observar únicamente los datos de interés y los aísla de otros datos en la base de datos. Puede existir cualquier número de vistas externas y ellos pueden ser completamente independientes o traslaparse entre sí.

El esquema conceptual se mapea a un esquema interno a nivel interno, el cual es el nivel de descripción más bajo de los datos en una base de datos. Este proporciona una interfaz al sistema de archivos del sistema operativo el cual es el responsable del acceso a la base de datos. El nivel interno tiene que ver con la especificación de qué elementos serán indexados, qué técnica de organización de archivos utilizar y como los datos se agrupan en el disco mediante "clusters" para mejorar su acceso.

En las Figuras 2.5, 2.6 y 2.7 se presenta la definición de los esquemas conceptual, interno y externo para las relaciones de la Figura 2.1.

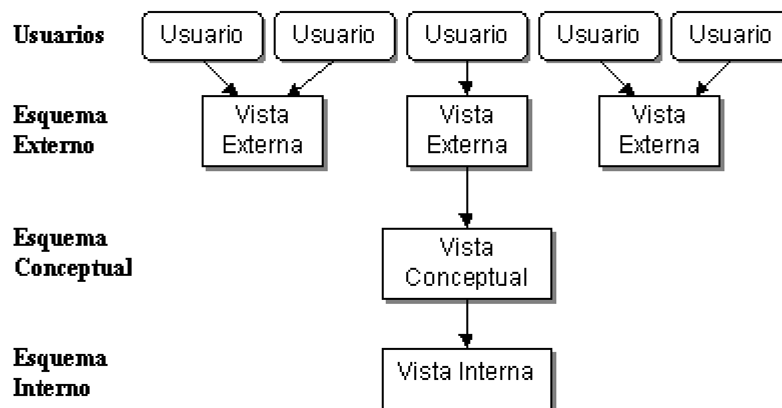
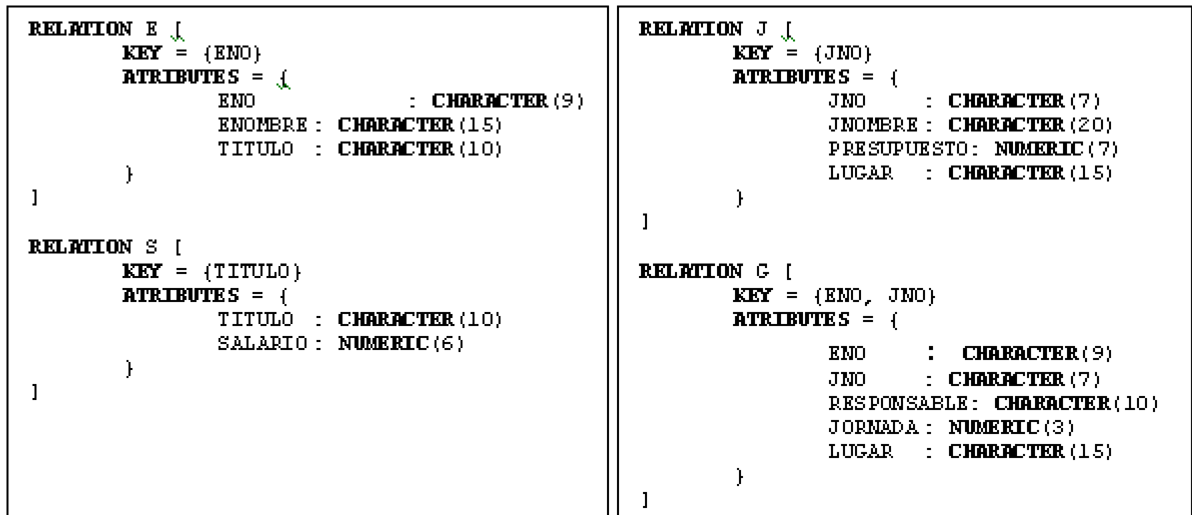



Figura 2.4. Arquitectura ANSI/SPARC de una base de datos.



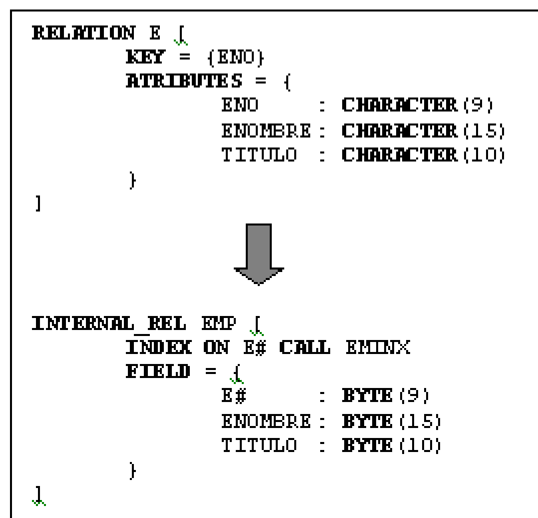
```
RELATION E [
        KEY = {ENO}
        ATRIBUTES = {
                ENO            : CHARACTER(9)
                ENOMBRE : CHARACTER(15)
                TITULO  : CHARACTER(10)
        }
]

RELATION S [
        KEY = {TITULO}
        ATRIBUTES = {
                TITULO  : CHARACTER(10)
                SALARIO : NUMERIC(6)
        }
]
```

```
RELATION J [
        KEY = {JNO}
        ATRIBUTES = {
                JNO     : CHARACTER(7)
                JNOMBRE : CHARACTER(20)
                PRESUPUESTO: NUMERIC(7)
                LUGAR   : CHARACTER(15)
        }
]

RELATION G [
        KEY = {ENO, JNO}
        ATRIBUTES = {
                ENO     :  CHARACTER(9)
                JNO     : CHARACTER(7)
                RESPONSABLE: CHARACTER(10)
                JORNADA : NUMERIC(3)
                LUGAR   : CHARACTER(15)
        }
]
```

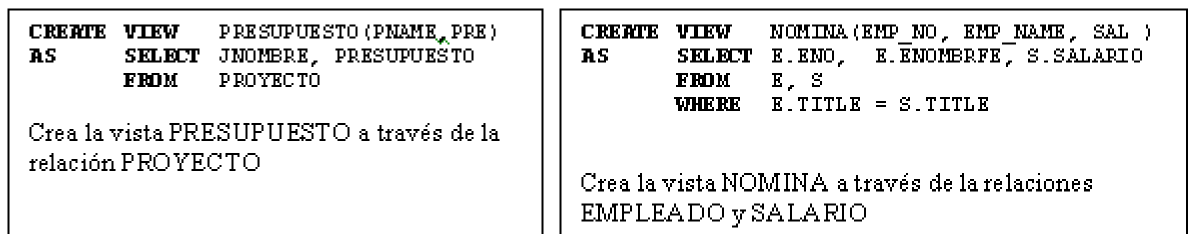
Figura 2.5. Vista conceptual de las relaciones E, S, J y G.



```
RELATION E [
        KEY = {ENO}
        ATRIBUTES = {
                ENO     : CHARACTER(9)
                ENOMBRE : CHARACTER(15)
                TITULO  : CHARACTER(10)
        }
]

INTERNAL_REL EMP [
        INDEX ON E# CALL EMINX
        FIELD = {
                E#      : BYTE(9)
                ENOMBRE : BYTE(15)
                TITULO  : BYTE(10)
        }
]
```

Figura 2.6. Definición de una vista interna a partir de la relación S.



```
CREATE  VIEW    PRESUPUESTO(PNAME,PRE)
AS      SELECT  JNOMBRE, PRESUPUESTO
        FROM    PROYECTO
```

Crea la vista PRESUPUESTO a través de la relación PROYECTO

```
CREATE  VIEW    NOMINA(EMP_NO, EMP_NAME, SAL )
AS      SELECT  E.ENO,  E.ENOMBRFE, S.SALARIO
        FROM    E, S
        WHERE   E.TITLE = S.TITLE
```

Crea la vista NOMINA a través de la relaciones EMPLEADO y SALARIO

Figura 2.7. Dos ejemplos de vistas externas.

Desafortunadamente, no existe un equivalente de una arquitectura estándar para sistemas de manejo de bases de datos distribuidas. La tecnología y prototipos de SMBDD se han desarrollado más o menos en forma independiente uno de otro y cada sistema ha adoptado su propia arquitectura.

Para definir un esquema de estandarización en bases de datos distribuidas se debe definir un ***modelo de referencia*** el cual sería un marco de trabajo conceptual cuyo propósito es dividir el trabajo de

estandarización en piezas manejables y mostrar a un nivel general como esas piezas se relacionan unas con otras. Para definir ese modelo de referencia se puede seguir uno de los siguientes tres enfoques:

1. **Basado en componentes**. Se definen las componentes del sistema junto con las relaciones entre ellas. Así, un SMBD consiste de un número de componentes, cada uno de los cuales proporciona alguna funcionalidad.
2. **Basado en funciones**. Se identifican las diferentes clases de usuarios junto con la funcionalidad que el sistema ofrecerá para cada clase. La especificación del sistema en esta categoría típicamente determina una estructura jerárquica para las clases de usuarios. La ventaja de este enfoque funcional es la claridad con la cual se especifican los objetivos del sistema. Sin embargo, este enfoque no proporciona una forma de alcanzar los objetivos.
3. **Basado en datos**. Se identifican los diferentes tipos de descripción de datos y se especifica un marco de trabajo arquitectural el cual define las unidades funcionales que realizarán y/o usarán los datos de acuerdo con las diferentes vistas. La ventaja de este enfoque es la importancia que asigna al manejo de datos. Este es un enfoque significativo para los SMBD dado que su propósito principal es manejar datos. Sin embargo, la desventaja de este enfoque es que es prácticamente imposible especificar un modelo arquitectural sin especificar los modelos para cada una de sus unidades funcionales. Este es el enfoque seguido por el modelo ANSI/SPARC.

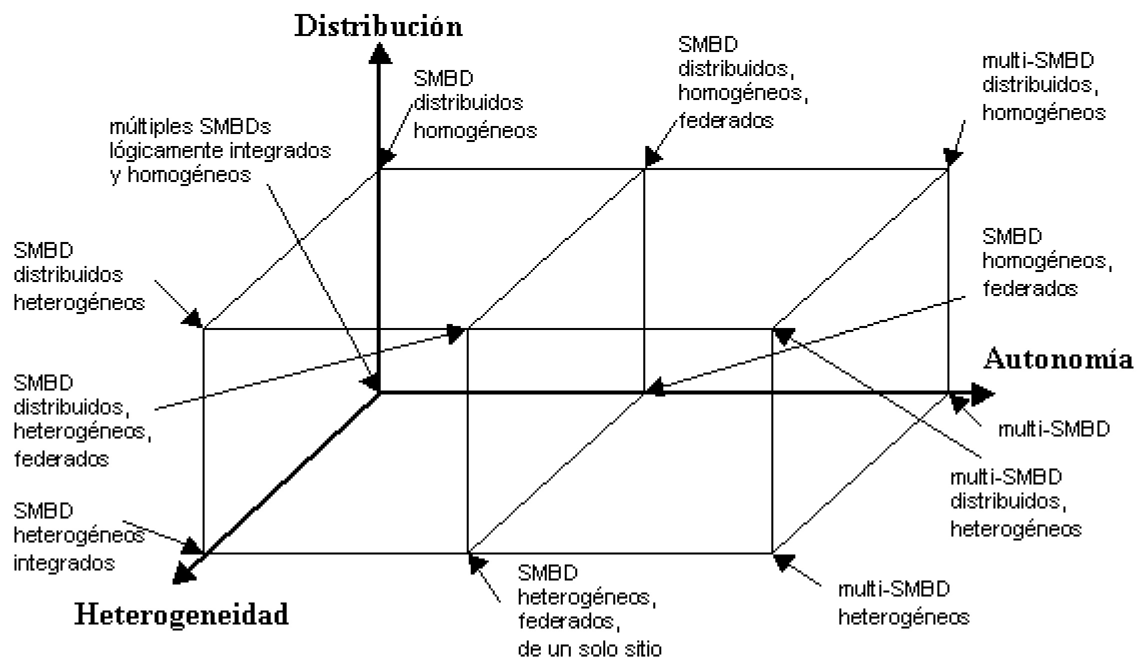



Figura 2.8. Dimensiones a considerar al integrar múltiples bases de datos.

## 2.3 ALTERNATIVAS PARA LA IMPLEMENTACION DE SMBD

En la Figura 2.8 se presentan las diferentes dimensiones (factores) que se deben considerar para la implementación de un sistema manejador de base de datos. Las dimensiones son tres:

1. **Distribución**. Determina si las componentes del sistema están localizadas en la misma computadora o no.
2. **Heterogeneidad**. La heterogeneidad se puede presentar a varios niveles: hardware, sistema de comunicaciones, sistema operativo o SMBD. Para el caso de SMBD heterogéneos ésta se puede presentar debido al modelo de datos, al lenguaje de consultas o a los algoritmos para manejo de transacciones.
3. **Autonomía**. La autonomía se puede presentar a diferentes niveles:

- **Autonomía de diseño**. La habilidad de un componente del SMBD para decidir cuestiones relacionadas a su propio diseño.
- **Autonomía de comunicación**. La habilidad de un componente del SMBD para decidir como y cuando comunicarse con otros SMBD.
- **Autonomía de ejecución**. La habilidad de un componente del SMBD para ejecutar operaciones locales de la manera que él quiera.

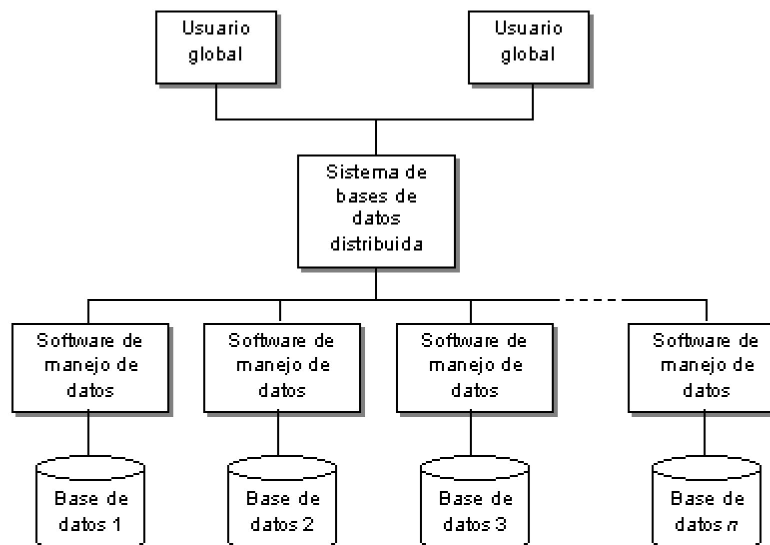



Figura 2.9. Arquitectura de un SMBDD homogéneo.

Desde el punto de vista funcional y de organización de datos, los sistemas de datos distribuidos están divididos en dos clases separadas, basados en dos filosofía totalmente diferentes y diseñados para satisfacer necesidades diferentes:

1. Sistemas de manejo de bases de datos distribuidos homogéneos
2. Sistemas de manejo de bases de datos distribuidos heterogéneos

Un ***SMBDD homogéneo*** tiene múltiples colecciones de datos; integra múltiples recursos de datos como se muestra en la Figura 2.9. Los sistemas homogéneos se parecen a un sistema centralizado, pero en lugar de almacenar todos los datos en un solo lugar, los datos se distribuyen en varios sitios comunicados por la red. No existen usuarios locales y todos ellos accesan la base de datos a través de una interfaz global. El esquema global es la unión de toda las descripciones de datos locales y las vistas de los usuarios se definen sobre el esquema global.

Para manejar los aspectos de la distribución, se deben agregar dos niveles a la arquitectura estándar ANSI-SPARC, como se muestra en la Figura 2.10. El **esquema de fragmentación** describe la forma en que las relaciones globales se dividen entre las bases de datos locales. La Figura 2.11 presenta el ejemplo de una relación, R, la cual se divide en cinco fragmentos. El **esquema de asignamiento** especifica el lugar en el cual cada fragmento es almacenado. De aquí, los fragmentos pueden migrar de un sitio a otro en respuesta a cambios en los patrones de acceso.

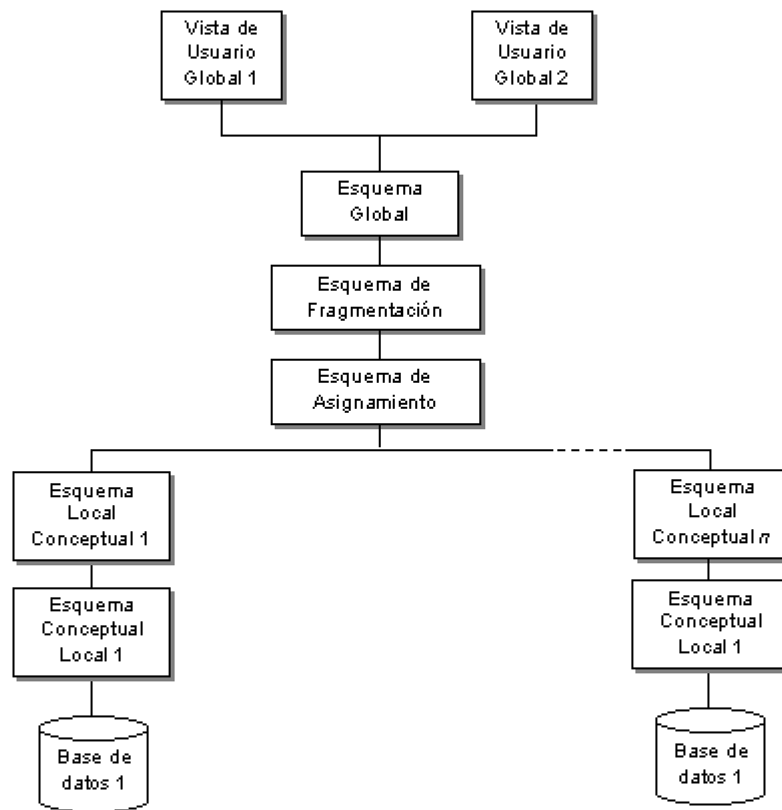



Figura 2.10. Arquitectura de los esquemas de un SMBDD homogéneo.

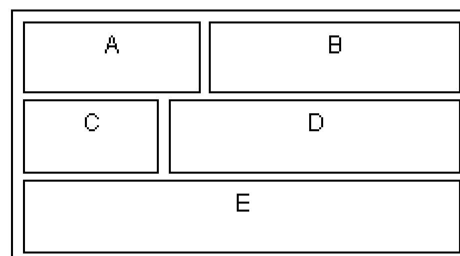



Figura 2.11. Fragmentación de una relación global.

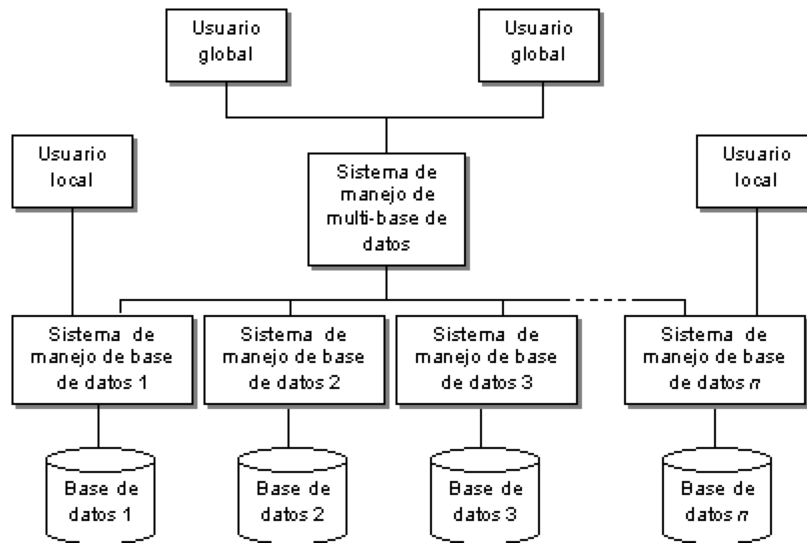



Figura 2.12. Arquitectura de un sistema multi-bases de datos.

La clase de ***sistemas heterogéneos*** es aquella caracterizada por manejar diferentes SMBD en los nodos locales. Una subclase importante dentro de esta clase es la de los **sistemas de manejo multi-bases de datos**. Un sistema multi-bases de datos (Smulti-BD) tiene múltiples SMBDs, que pueden ser de tipos diferentes, y múltiples bases de datos existentes. La integración de todos ellos se realiza mediante subsistemas de software. La arquitectura general de tales sistemas se presenta en la Figura 2.12. En contraste con los sistemas homogéneos, existen usuarios locales y globales. Los usuarios locales accesan sus bases de datos locales sin verse afectados por la presencia del Smulti-BD.

En algunas ocasiones es importante caracterizar a los sistemas de bases de datos distribuidas por la forma en que se organizan sus componentes. En la Figura 2.13 se presenta la arquitectura basada en componentes de un SMBD distribuido. Consiste de dos partes fundamentales, el procesador de usuario y el procesador de datos. El procesador de usuario se encarga de procesar las solicitudes del usuario, por tanto, utiliza el esquema externo del usuario y el esquema conceptual global. Así también, utiliza un **diccionario de datos global**. El procesador de usuario consiste de cuatro partes: un manejador de la interfaz con el usuario, un controlador semántico de datos, un optimizador global de consultas y un supervisor de la ejecución global. El procesador de datos existe en cada nodo de la base de datos distribuida. Utiliza un esquema local conceptual y un esquema local interno. El procesador de datos consiste de tres partes: un procesador de consultas locales, un manejador de recuperación de fallas locales y un procesador de soporte para tiempo de ejecución.

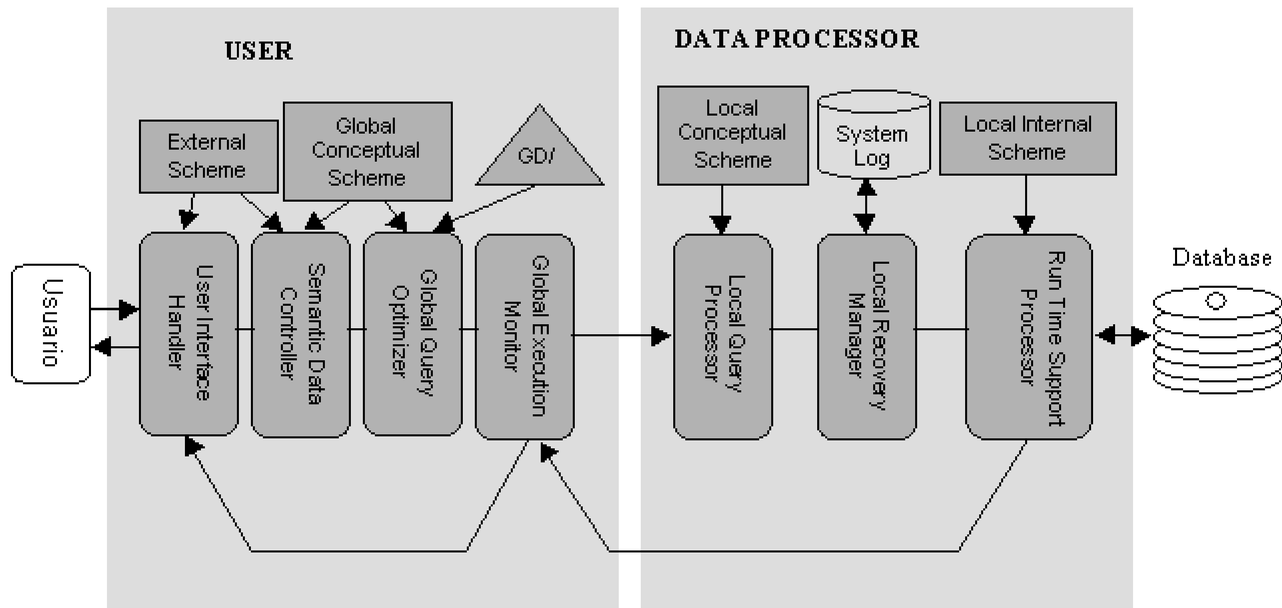



Figura 2.13. Arquitectura basada en componentes de un SMBD distribuido.

En la Figura 2.14, se presenta la arquitectura basada en componentes de un sistema multi-bases de datos. Consiste un sistema de manejo de bases datos para usuarios globales y de sistemas de manejo de bases de datos para usuarios locales. Las solicitudes globales pasan al procesador global el cual consiste de un procesador de transacciones, una interfaz de usuario, un procesador de consultas, un optimizador de consultas, un esquema y un administrador de recuperación de fallas, todos ellos actuando de manera **global**.

En cada sitio existe un SMBD completo el cual consiste de la interfaz con el usuario, el procesador y optimizador de consultas, el manejador de transacciones, el despachador de operaciones, el manejador de recuperación de fallas y el sistema de soporte para tiempo de ejecución, todos ellos actuando de manera **local**. Para comunicar el sistema global con los sistemas locales se define una interfaz común entre componentes mediante la cual, las operaciones globales se convierten en una o varias acciones locales.

El manejo de directorio de datos es de una importancia mayor en bases de datos distribuidas. Por un lado, puede haber directorios locales o un solo directorio global. Por otra parte, su manejo puede ser local o distribuido. Finalmente, desde otro punto de vista el directorio puede ser replicado o no replicado. Como se puede ver en la Figura 2.15, existen combinaciones, en estas tres dimensiones, que no tienen mayor relevancia. Sin embargo, en varios de los vértices del cubo en tres dimensiones aparecen las combinaciones importantes para bases de datos distribuidas.

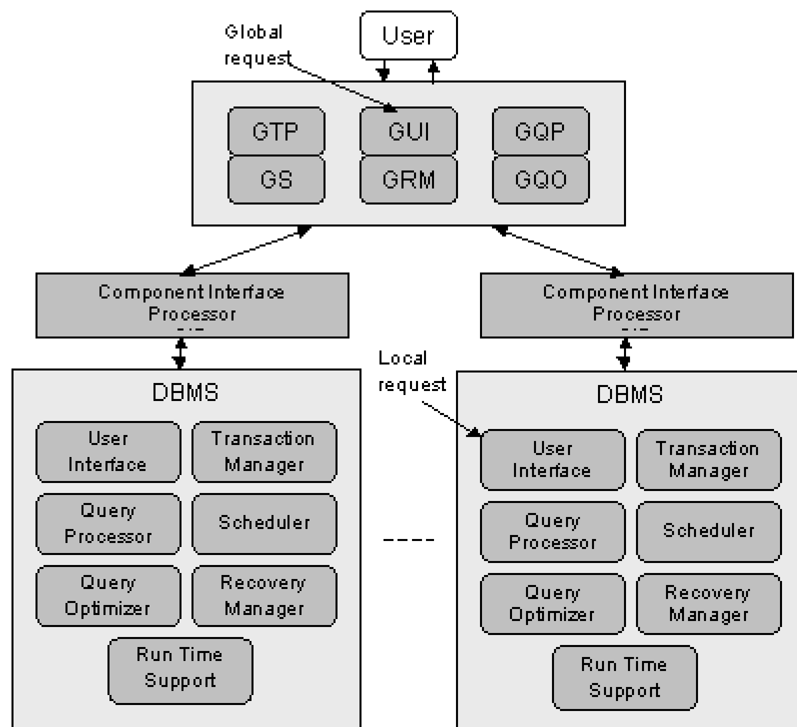



Figura 2.14. Arquitectura basada en componentes de un sistema multi-bases de datos.

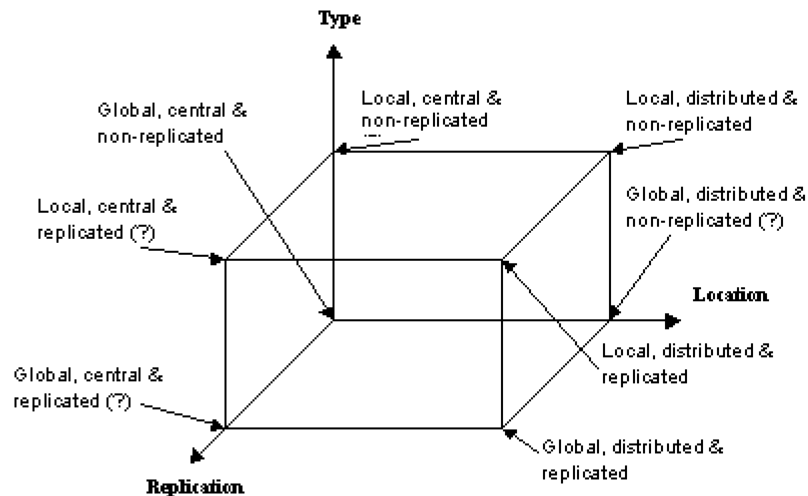



Figura 2.15. Manejo del directorio de datos en bases de datos distribuidas.

# CAPITULO 3. DISEÑO DE BASE DE DATOS DISTRIBUIDA

En el presente capítulo se mostrará los aspectos importantes referentes al diseño de una base de datos distribuida. Se revisará el problema de fragmentación de los datos así como la transparencia que un sistema de datos distribuidos debe guardar respecto a la vista del usuario. Se presentarán los algoritmos para fragmentación horizontal, fragmentación horizontal derivada y fragmentación vertical. En la parte final de este capítulo se discute el problema de asignamiento de fragmentos.

## 3.1 El problema de diseño

El problema de diseño de bases de datos distribuidos se refiere, en general, a hacer decisiones acerca de la ubicación de *datos* y *programas* a través de los diferentes sitios de una red de computadoras. Este problema debería estar relacionado al diseño de la misma red de computadoras. Sin embargo, en estas notas únicamente el diseño de la base de datos se toma en cuenta. La decisión de donde colocar a las aplicaciones tiene que ver tanto con el software del SMBDD como con las aplicaciones que se van a ejecutar sobre la base de datos.

El diseño de las bases de datos centralizadas contempla los dos puntos siguientes:

1. Diseño del **"esquema conceptual"** el cual describe la base de datos integrada (esto es, todos los datos que son utilizados por las aplicaciones que tienen acceso a las bases de datos).

2. Diseño **"físico de la base de datos"**, esto es, mapear el esquema conceptual a las áreas de almacenamiento y determinar los métodos de acceso a las bases de datos.

En el caso de las bases de datos distribuidas se tienen que considerar los dos problemas siguientes:

3. Diseño de la **fragmentación**, este se determina por la forma en que las relaciones globales se subdividen en fragmentos horizontales, verticales o mixtos.

4. Diseño de la **asignación de los fragmentos**, esto se determina en la forma en que los fragmentos se mapean a las imágenes físicas, en esta forma, también se determina la solicitud de fragmentos.

### Objetivos del Diseño de la Distribución de los Datos.

En el diseño de la distribución de los datos, se deben de tomar en cuenta los siguientes objetivos:

- **Procesamiento local.** La distribución de los datos, para maximizar el procesamiento local corresponde al principio simple de colocar los datos tan cerca como sea posible de las aplicaciones que los utilizan. Se puede realizar el diseño de la distribución de los datos para maximizar el procesamiento local agregando el número de referencias locales y remotas que le corresponden a cada fragmentación candidata y la localización del fragmento, que de esta forma se seleccione la mejor solución de ellas.
- **Distribución de la carga de trabajo.** La distribución de la carga de trabajo sobre los sitios, es una característica importante de los sistemas de cómputo distribuidos. Esta distribución de la carga se realiza para tomar ventaja de las diferentes características (potenciales) o utilizaciones de las computadoras de cada sitio, y maximizar el grado de ejecución de paralelismo de las aplicaciones. Sin embargo, la distribución de la carga de trabajo podría afectar negativamente el procesamiento local deseado.

- **Costo de almacenamiento y disponibilidad.** La distribución de la base de datos refleja el costo y disponibilidad del almacenamiento en diferentes sitios. Para esto, es posible tener sitios especializados en la red para el almacenamiento de datos. Sin embargo el costo de almacenamiento de datos no es tan relevante si éste se compara con el del CPU, I/O y costos de transmisión de las aplicaciones.

### 3.2 Enfoques al problema de diseño de bases de datos distribuidas

Existen dos estrategias generales para abordar el problema de diseño de bases de datos distribuidas:

1. **El enfoque de arriba hacia abajo (top-down)**. Este enfoque es más apropiado para aplicaciones nuevas y para sistemas homogéneos. Consiste en partir desde el análisis de requerimientos para definir el diseño conceptual y las vistas de usuario. A partir de ellas se define un esquema conceptual global y los esquemas externos necesarios. Se prosigue con el diseño de la fragmentación de la base de datos, y de aquí se continúa con la localización de los fragmentos en los sitios, creando las imágenes físicas. Esta aproximación se completa ejecutando, en cada sitio, "el diseño físico" de los datos, que se localizan en éste. En la Figura 3.1 se presenta un diagrama con la estructura general del enfoque top-down.
2. **El diseño de abajo hacia arriba (bottom-up)**. Se utiliza particularmente a partir de bases de datos existentes, generando con esto bases de datos distribuidas. En forma resumida, el diseño bottom-up de una base de datos distribuida requiere de la selección de un modelo de bases de datos común para describir el esquema global de la base de datos. Esto se debe es posible que se utilicen diferentes SMBD. Después se hace la traducción de cada esquema local en el modelo de datos común y finalmente se hace la integración del esquema local en un esquema global común.

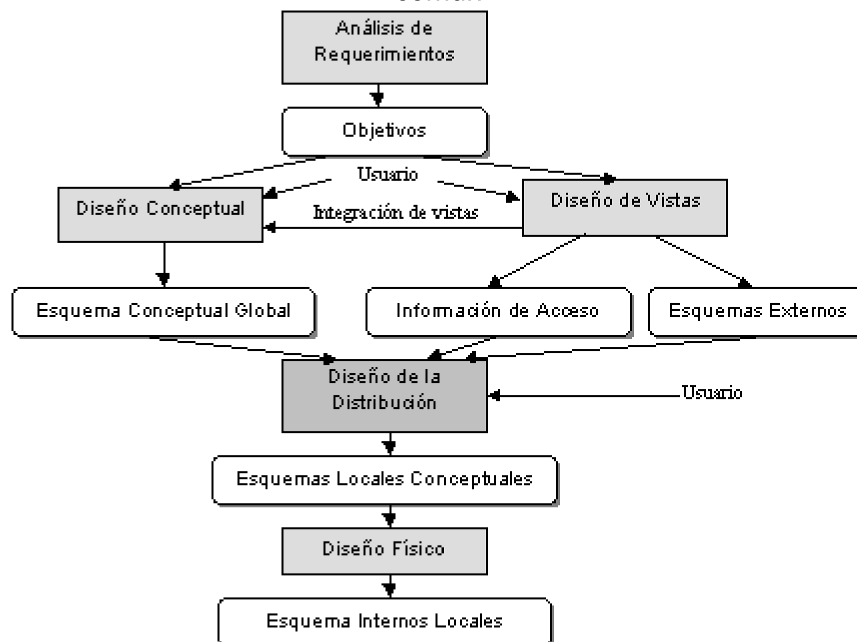



Figura 3.1. El enfoque top-down para el diseño de bases de datos distribuidas.

El diseño de una base de datos distribuida, cualquiera sea el enfoque que se siga, debe responder satisfactoriamente las siguientes preguntas:

- ¿Por qué hacer una fragmentación de datos?
- ¿Cómo realizar la fragmentación?
- ¿Qué tanto se debe fragmentar?

- ¿Cómo probar la validez de una fragmentación?
- ¿Cómo realizar el asignamiento de fragmentos?
- ¿Cómo considerar los requerimientos de la información?

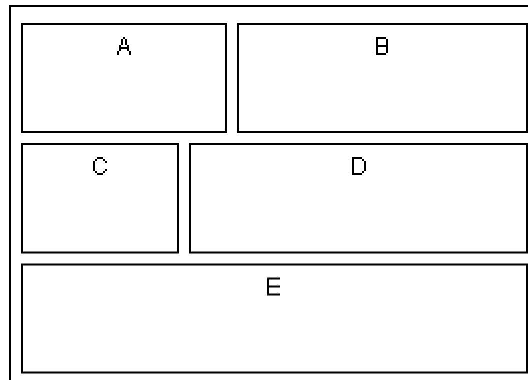



Figura 3.2. El problema de fragmentación de relaciones.

### 3.3 El problema de fragmentación

El problema de fragmentación se refiere al particionamiento de la información para distribuir cada parte a los diferentes sitios de la red, como se observa en la Figura 3.2. Inmediatamente aparece la siguiente pregunta: ¿cuál es la unidad razonable de distribución?. Se puede considerar que una relación completa es lo adecuado ya que las vistas de usuario son subconjuntos de las relaciones. Sin embargo, el uso completo de relaciones no favorece las cuestiones de eficiencia sobre todo aquellas relacionadas con el procesamiento de consultas.

La otra posibilidad es usar fragmentos de relaciones (sub-relaciones) lo cual favorece la ejecución concurrente de varias transacciones que accesan porciones diferentes de una relación. Sin embargo, el uso de sub-relaciones también presenta inconvenientes. Por ejemplo, las vistas de usuario que no se pueden definir sobre un solo fragmento necesitarán un procesamiento adicional a fin de localizar todos los fragmentos de una vista. Aunado a esto, el control semántico de datos es mucho más complejo ya que, por ejemplo, el manejo de llaves únicas requiere considerar todos los fragmentos en los que se distribuyen todos los registros de la relación. En resumen, el objetivo de la fragmentación es encontrar un nivel de particionamiento adecuado en el rango que va desde tuplas o atributos hasta relaciones completas (ver Figura 3.3).

**Ejemplo 3.1**. Considere la relación J del ejemplo visto en el capítulo 2.

J:

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J1 | Instrumentación | 150000 | Monterrey |
| J2 | Desarrollo de bases de datos | 135000 | México |
| J3 | CAD/CAM | 250000 | Puebla |
| J4 | Mantenimiento | 310000 | México |
| J5 | CAD/CAM | 500000 | Guadalajara |

La relación J se puede fragmentar horizontalmente produciendo los siguientes fragmentos.

J1: proyectos con presupuesto menor que $200,000

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J1 | Instrumentación | 150000 | Monterrey |
| J2 | Desarrollo de bases de datos | 135000 | México |

J2: proyectos con presupuesto mayor que o igual a $200,000

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J3 | CAD/CAM | 250000 | Puebla |
| J4 | Mantenimiento | 310000 | México |
| J5 | CAD/CAM | 500000 | Guadalajara |

..

**Ejemplo 3.2**. La relación J del ejemplo anterior se puede fragmentar verticalmente produciendo los siguientes fragmentos:

J1: información acerca de presupuestos de proyectos

| JNO | PRESUPUESTO |
|---|---|
| J1 | 150000 |
| J2 | 135000 |
| J3 | 250000 |
| J4 | 310000 |
| J5 | 500000 |

J2: información acerca de los nombres y ubicaciones de proyectos

| JNO | JNOMBRE | LUGAR |
|---|---|---|
| J1 | Instrumentación | Monterrey |
| J2 | Desarrollo de bases de datos | México |
| J3 | CAD/CAM | Puebla |
| J4 | Mantenimiento | México |
| J5 | CAD/CAM | Guadalajara |

..

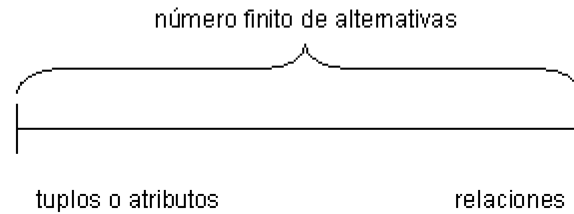



Figura 3.3. El grado de fragmentación.

**Correctitud de una fragmentación**Al realizar la fragmentación de una relación se deben satisfacer las siguientes condiciones para garantizar la correctitud de la misma:

1. **Condición de completitud**. La descomposición de una relación $R$ en los fragmentos $R_1$, $R_2$, ..., $R_n$ es completa si y solamente si cada elemento de datos en $R$ se encuentra en algún de los $R_i$.
2. **Condición de Reconstrucción**. Si la relación $R$ se descompone en los fragmentos $R_1$, $R_2$, ..., $R_n$, entonces debe existir algún operador relacional Ñ , tal que, $R$ = Ñ $_{1£ i£ n}$ $R_i$
3. **Condición de Fragmentos Disjuntos**. Si la relación $R$ se descompone en los fragmentos $R_1$, $R_2$, ..., $R_n$, y el dato $d_i$ está en $R_j$, entonces, no debe estar en ningún otro fragmento $R_k$ (*k¹ j*).

**Alternativas sobre replicación para el asignamiento de fragmentos**

La replicación de información es de utilidad para obtener un mejor rendimiento y para ofrecer un mayor grado de confiabilidad (tolerancia a fallas). La replicación se complica cuando es necesario hacer actualizaciones a las copias múltiples de un dato. Por tanto, respecto a la replicación, en el asignamiento de fragmentos se tienen tres estrategias:

1. No soportar replicación. Cada fragmento reside en un solo sitio.
2. Soportar replicación completa. Cada fragmento en cada uno de los sitios.
3. Soportar replicación parcial. Cada fragmento en algunos de los sitios.

Como regla general se debe considerar que la replicación de fragmentos es de utilidad cuando el número de consultas de solo lectura es (mucho) mayor que el número de consultas para actualizaciones. En la Tabla 3.1 se comparan la complejidad de implementar o tomar ventaja de las diferentes alternativas de replicación, respecto de los diferentes aspectos importantes en bases de datos distribuidas.

| | **Replicación Completa** | **Replicación Parcial** | **Particionamiento** |
|---|---|---|---|
| **Procesamiento de Consultas** | Fácil | Moderado | Moderado |
| **Manejo de Directorios** | Fácil o no existente | Moderado | Moderado |
| **Control de Concurrencia** | Moderado | Difícil | Fácil |
| **Confiabilidad** | Muy alto | Alto | Bajo |
| **Realidad** | Aplicación posible | Realista | Aplicación posible |

Tabla 3.1. Comparación de las estrategias de replicación de fragmentos.

**Requerimientos de información**

Con el fin de realizar una fragmentación adecuada es necesario proporcionar información que ayude a realizarla. Esta información normalmente debe ser proporcionada por el usuario y tiene que ver con cuatro tipos:

1. Información sobre el significado de los datos
2. Información sobre las aplicaciones que los usan
3. Información acerca de la red de comunicaciones
4. Información acerca de los sistemas de cómputo

## 3.4. Tipos de fragmentación de datos

Existen tres tipos de fragmentación:

1. Fragmentación horizontal
2. Fragmentación vertical
3. Fragmentación híbrida

En las siguientes secciones revisaremos de manera informal cada uno de los tipos mencionados. Más adelante, se presentará de manera más formal la construcción de los diferentes tipos de fragmentación.

### 3.4.1 Fragmentación horizontal primaria

Consiste del particionamiento en tuplas de una relación global en subconjuntos, donde cada subconjunto puede contener datos que tienen propiedades comunes y se puede definir expresando cada fragmento como una operación de selección sobre la relación global.

**Ejemplo 3.3**. Considere la relación global

SUPPLIER( SNUM, NAME, CITY )

entonces, la fragmentación horizontal puede ser definida como:

SUPPLIER1 = SLcity == "SF"SUPPLIER

SUPPLIER1 = SLcity == "LA"SUPPLIER

1. Esta fragmentación satisface la condición de completes si "SF" y "LA" son solamente los únicos valores posibles del atributo CITY.

2. La condición de reconstrucción se logra con:

SUPPLIER = SUPPLIER1 union SUPPLIER2

3. La condición de disjuntos se cumple claramente en este ejemplo.

..

### 3.4.2 Fragmentación horizontal derivada

La fragmentación derivada horizontal se define partiendo de una fragmentación horizontal.

En esta operación se requiere de Semi-junta (Semi-Join) el cual nos sirve para derivar las tuplas o registros de dos relaciones.

**Ejemplo 3.4**. Las siguientes relaciones definen una fragmentación horizontal derivada de la relación SUPPLY.

SUPPLY1 = SUPPLYSJsnum == snumSUPPLIER1

SUPPLY2 = SUPPLYSJsnum == snumSUPPLIER2

..

### 3.4.3 Fragmentación vertical

La fragmentación vertical es la subdivisión de atributos en grupos. Los fragmentos se obtienen proyectando la relación global sobre cada grupo. La fragmentación es correcta si cada atributo se mapea en al menos un atributo del fragmento.

**Ejemplo 3.5**. Considere la siguiente relación global:

EMP( empnum, name, sal, tax, mgrnum, depnum )

una fragmentación vertical de esta relación puede ser definida como:

EMP1 = PJempnum, name, mgrnum, depnum EMP

EMP2 = PJempnum, sal, tax EMP

la reconstrucción de la relación EMP puede ser obtenida como:

EMP = EMP1 (JN empnum) EMP2 porque empnum es una clave de EMP

..

### 3.4.4 Fragmentación híbrida

En la que respecto a la fragmentación híbrida, esta consiste en aplicar la fragmentación vertical seguida de la fragmentación horizontal o viceversa.

**Ejemplo 3.6**. Considere la relación global

EMP( empnum, name, sal, tax, mrgnum, depnum )

Las siguientes son para obtener una fragmentación mixta, aplicando la vertical seguida de la horizontal:

EMP1 = SL depnum <= 10 PJempnum, name, mgrnum, depnum EMP

EMP2 = SL 10 < depnum <= 20 PJempnum, name, mgrnum, depnum EMP

EMP3 = SL depnum > 20 PJempnum, name, mgrnum, depnum EMP

EMP4 = PJ empnum, name, sal, tax EMP

La reconstrucción de la relación EMP es definida por la siguiente expresión:

EMP = UN(EMP1, EMP2, EMP3)JNempnum = empnumPJempnum, sal, tax EMP4

Finalmente, como otro ejemplo considere el siguiente esquema global

EMP(EMPNUM, NAME, SAL, TAX, MGRNUM, DEPNUM)

DEP(DEPNUM, NAME, AREA, MGRNUM)

SUPPLIER(SNUM, NAME, CITY)

SUPPLY(SNUM, PNUM, DEPNUM, QUAN)

Después de aplicar una fragmentación mixta se obtiene el siguiente esquema fragmentado

EMP1 = Sldepnum <= 10 PJempnum, name, mgrnum, depnum (EMP)

EMP2 = SL 10 < depnum <= 20 PJempnum, name, mgrnum, depnum (EMP)

EMP3 = SL depnum > 20 PJempnum, name, mgrnum, depnum (EMP)

EMP4 = PJ empnum, name, sal, tax (EMP)

DEP1 = SL depnum <= 10 (DEP)

DEP2 = SL 10 < depnum <= 20 (DEP)

DEP3 = SL depnum > 20 (DEP)

SUPPLIER1 = SL city == "SF" (SUPPLIER)

SUPPLIER2 = SL city == "LA" (SUPPLIER)

SUPPLY1 = SUPPLYSJsnum == snumSUPPLIER1

SUPPLY2 = SUPPLYSJsnum == snumSUPPLIER2

### 3.5 Fragmentación horizontal

En las siguientes secciones revisaremos de manera más formal la forma de construir los diferentes tipos de fragmentación.

La fragmentación horizontal primaria de una relación se obtiene usando predicados que están definidos en esa relación. La fragmentación horizontal derivada, por otra parte, es el particionamiento de una relación como resultado de predicados que se definen en otra relación.

Para poder construir una fragmentación, es necesario proporcionar información acerca de la base de datos y acerca de las aplicaciones que las utilizan. En primer término, es necesario proporcionar la información acerca del esquema conceptual global. En este sentido es importante dar información acerca de las relaciones que componen a la base de datos, la cardinalidad de cada relación y las dependencias entre relaciones. Por ejemplo, en la Figura 3.4 se presenta un diagrama mostrando el esquema conceptual de la base de datos de ejemplo del capítulo 2.

En segundo lugar se debe proporcionar información acerca de la aplicación que utiliza la base de datos. Este tipo de información es cuantitativa y consiste de los predicados usados en las consultas de usuario.

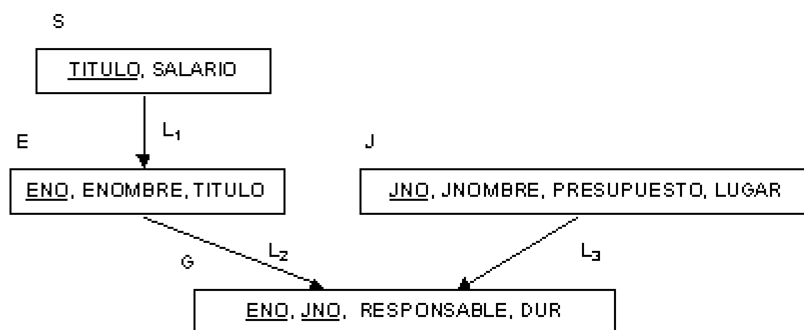



Figura 3.4. Esquema conceptual de la base de datos de ejemplo del capítulo 2.

Dada una relación $R(A_1, A_2, \ldots, A_n)$, donde $A_i$ es un atributo definido sobre el dominio $D_i$, un ***predicado simple*** $p_j$ definido en $R$ tiene la forma

$p_j$: $A_i$ q *Valor*

donde q Î { =, <, ¹ , £ , >, ³ } y *Valor* Î $D_i$. Para la relación $R$ se define un conjunto de predicados simples como $Pr = \{ p_1, p_2, \ldots, p_m \}$.

**Ejemplo 3.7**. Las siguientes expresiones se consideran como predicados simples.

JNOMBRE = "Mantenimiento"

PRESUPUESO < 200000

..

Dado la relación $R$ y el conjunto de predicados simples $Pr = \{ p_1, p_2, \ldots, p_m \}$, se define el conjunto de ***predicados minitérmino*** como $M = \{ m_1, m_2, \ldots, m_r \}$ como

*M* = { $m_i$ | $m_i$ = Ù $_{pj\ \hat{I}\ Pr}$ $p_j$*}, 1 £ *j* £ *m*, 1 £ *i* £ *z*

donde, $p_j$* = $p_j$ o $p_j$* = Ø ($p_j$).

**Ejemplo 3.8**. Los siguientes son minitérminos de la relación J.

$m_1$: JNOMBRE == "Mantenimiento" Ù Presupuesto £ 200000

$m_2$: **NOT**( JNOMBRE == "Mantenimiento") Ù Presupuesto £ 200000

$m_3$: JNOMBRE == "Mantenimiento" Ù **NOT** (Presupuesto £ 200000)

$m_4$: **NOT**( JNOMBRE == "Mantenimiento") Ù **NOT**(Presupuesto £ 200000)

..

En términos de la información cuantitativa acerca de las aplicaciones de usuario, se necesita tener dos conjuntos de datos:

1. La **selectividad de los minitérminos**: Denotada como *sel*( $m_i$ ), se refiere al número de tuplos de la relación que serán accesadas por una consulta de usuario especificada de acuerdo a un predicado minitérmino dado.
2. La **frecuencia de acceso**: Denotada como *acc*( $q_i$ ), se refiere a la frecuencia con la cual una consulta de usuario $q_i$ es accesada en un periodo de tiempo. Note que las frecuencias de acceso de minitérminos se pueden determinar a partir de las frecuencias de consultas. La frecuencia de acceso de un minitérmino se denota como *acc*( $m_i$ ).

Una fragmentación horizontal primaria se define por una operación de selección en las relaciones propietarias de un esquema de la base de datos. Por tanto, dada una relación R, su fragmentación horizontal está dada por

$R_j$ = s $_{Fj}$ ($R$), 1 £ $j$ £ $w$

donde, $F_j$ es una fórmula de selección, la cual es preferiblemente un predicado minitérmino. Por lo tanto, un fragmento horizontal $R_i$ de una relación $R$ consiste de todos los tuplos de $R$ que satisfacen un predicado minitérmino $m_i$. Lo anterior implica que dado un conjunto de predicados minitérmino $M$, existen tantos fragmentos horizontales de $R$ como minitérminos existan. El conjunto de fragmentos horizontales también se entiende como los ***fragmentos minitérminos***.

Es necesario desarrollar un algoritmo que tome como entrada una relación $R$ y el conjunto de predicados simples $Pr$ y proporcione como resultado el conjunto de fragmentos de $R$ = { $R_1$, $R_2$, …, $R_m$ } el cual obedece las reglas de fragmentación. Un aspecto importante del conjunto de predicados es que debe ser ***completo*** y ***minimal***.

Un conjunto de predicados simples $Pr$ se dice que es ***completo*** si y solo si los accesos a los tuplos de los fragmentos minitérminos definidos en $Pr$ requieren que dos tuplos del mismo fragmento tengan la misma probabilidad de ser accesados por cualquier aplicación.

**Ejemplo 3.9** Considere que la relación J[JNO, JNOMBRE, PRESUPUESTO, LUGAR] tiene dos consultas definidas en ella:

Encontrar todos los presupuestos de los proyectos en cada lugar (1)

Encontrar proyectos con presupuestos menores a $200000. (2)

De acuerdo a (1),

*Pr* = { LUGAR = "México", LUGAR = "Puebla", LUGAR = "Guadalajara", LUGAR = "Monterrrey" }

no es completa con respecto a (2) dado que algunos de los tuplos dentro de cada $J_i$ tienen una probabilidad mayor de ser accesados por la segunda consulta. Si se modifica *Pr* como

*Pr* = { LUGAR = "México", LUGAR = "Puebla", LUGAR = "Guadalajara", LUGAR = "Monterrrey",

PRESUPUESTO £ 2000000, PRESUPUESTO > 200000 }

entonces, Pr es completo.

..

De manera intuitiva se puede ver que si un predicado influye en la fragmentación, esto es, causa que un fragmento *f* se fragmente aún más digamos en $f_i$ y $f_j$, entonces habría una consulta que accese $f_i$ y $f_j$ de manera diferente. En otras palabras, un predicado debe ser relevante en determinar una fragmentación. Si todos los predicados de un conjunto *Pr* son relevantes, entonces *Pr* es ***mínimo***.

La definición formal de relevancia es la siguiente. Sean $m_i$ y $m_j$ dos predicados minitérminos definidos exactamente igual excepto que $m_i$ contiene a $p_i$ y $m_j$ contiene a $p_j$. También, sean $f_i$ y $f_j$ los dos fragmentos definidos de acuerdo a $m_i$ y $m_j$, respectivamente. Entonces, $p_i$ es ***relevante*** si y solo si

$acc(m_i)/card(f_i)$ ¹ $acc(m_j)/card(f_j)$

Por ejemplo, el conjunto *Pr* definido arriba es mínimo y completo. Sin embargo, si se le agrega el predicado JNOMBRE = "Instrumentación", entonces, *Pr* no es mínimo.

El algoritmo siguiente llamado COM_MIN genera un conjunto completo y mínimo de predicados *Pr'* dado un conjunto de predicados simple *Pr*. Por brevedad durante el algoritmo se utiliza la siguiente regla:

*Regla* 1: regla fundamental de completes y minimalidad, la cual afirma que una relación o fragmento es particionado en al menos dos partes las cuales se accesan en forma diferente por al menos una consulta de usuario.

**Algoritmo 3.1** COM_MIN

Entrada: una relación *R* y un conjunto de predicados simples *Pr*

Salida: un conjunto completo y mínimo de predicados simples *Pr'* para *Pr*.

1. Iniciación:

    - Encontrar un $p_i$ Î *Pr* tal que $p_i$ particiona a *R* de acuerdo a la regla 1.
    - Hacer *Pr'* = $p_i$; *Pr* ¬ *Pr* – $p_i$; *F* ¬ $f_i$

1. Iterativamente agregar predicados a *Pr'* hasta que sea completo

   - Encontrar un $p_i$ Î *Pr* tal que $p_i$ particiona algún $f_k$ de *Pr'* de acuerdo a la regla 1.
   - Hacer *Pr'* = *Pr'* U $p_i$; *Pr* ¬ *Pr* – $p_i$; *F* ¬ *F* U $f_i$
   - Si $ $p_k$ Î *Pr'* el cual es no relevante, entonces,

     Hacer *Pr'* = *Pr'* - $p_k$; *F* ¬ *F* - $f_k$

El algoritmo empieza encontrando un predicado que es relevante y que particiona la relación de entrada. Después, agrega de manera iterativa predicados a este conjunto, asegurando minimalidad en cada paso. Por lo tanto, al final el conjunto *Pr'* es tanto completo como mínimo.

El segundo paso en el proceso de diseño de fragmentación horizontal primaria es derivar el conjunto de predicados minitérminos que pueden ser definidos en los predicados del conjunto *Pr'*. Esos minitérminos definen los fragmentos que serán usados como candidatos en el paso de asignamiento.

El algoritmo de fragmentación horizontal primaria, llamado PHORIZONTAL, se presenta a continuación. La entrada al algoritmo es una relación $R_i$ la cual es sometida a fragmentación horizontal primaria, y $Pr_i$, el cual es el conjunto de predicados simples que han sido determinados de acuerdo a las consultas definidas en la relación $R_i$.

**Algoritmo 3.2** PHORIZONTAL

Entrada: Una relación *R* y un conjunto de predicados simples *Pr*.

Salida: Un conjunto de predicados minitérminos, *M*, de acuerdo a los cuales la relación *R* será fragmentada.

1. *Pr'* ¬ COM_MIN( *R*, *Pr* )
2. determinar el conjunto *M* de predicados minitérminos
3. determinar el conjunto *I* de implicaciones entre $p_i$ Î *Pr*
4. eliminar minitérminos contradictorios a partir de *M*

**Ejemplo 3.10**. Para la relación S la consulta o aplicación es verificar la información del salario y determinar incrementos. Suponga además que los registros de empleados se mantienen en dos lugares y, por tanto, la aplicación o consulta se ejecuta en dos lugares.

Los predicados simples que serían usados para particionar la relación S son:

$p_1$ : SAL £ 30000

$p_2$ : SAL > 30000

Al aplicar el algoritmo COM_MIN se verifica que *Pr* = { $P_1$, $P_2$ } se completo y minimal, *Pr'* = *Pr*. Se pueden formar los siguientes predicados minitérminos como miembros de *M*:

$m_1$: (SAL £ 30000) Ù (SAL > 30000)

$m_2$: (SAL £ 30000) Ù NOT (SAL > 30000)

$m_3$: NOT (SAL £ 30000) Ù (SAL > 30000)

$m_4$: NOT (SAL £ 30000) Ù NOT (SAL > 30000)

Asumiendo que el dominio de SALARIO se puede partir en dos, como se sugiere *Pr* $p_1$ y $p_2$, las siguientes implicaciones son obvias:

$i_1$: (SAL £ 30000) Þ NOT (SAL > 30000)

$i_2$: NOT (SAL £ 30000) Þ (SAL > 30000)

$i_3$: (SAL > 30000) Þ NOT (SAL £ 30000)

$i_4$: NOT (SAL > 30000) Þ (SAL £ 30000)

De acuerdo a $i_1$, $m_1$ es contradictorio; de acuerdo a $i_2$, $m_4$ es contradictorio. Por lo tanto, nos quedamos con $M$ = { $m_2$, $m_3$ }. Por tanto, se definen los dos fragmentos $F_s$ = { $S_1$, $S_2$ } de acuerdo a $M$.

$S_1$

| TITULO | SALARIO |
|---|---|
| Ingeniero Mecánico | 27000 |
| Programador | 24000 |

$S_1$

| TITULO | SALARIO |
|---|---|
| Ingeniero Eléctrico | 40000 |
| Analista de Sistemas | 34000 |

..

**Ejemplo 3.11**. Para la relación J la consulta es encontrar el nombre y presupuesto de proyectos dados por su número. Esta consulta es realizada en tres lugares. El acceso a la información de proyecto se realiza de acuerdo a su presupuesto; un lugar accesa presupuesto £ 200000 y el otro accesa presupuesto > 200000.

Los predicados simples para la primera consulta serían:

$p_1$ : LUGAR = "México"

$p_2$ : LUGAR = "Monterrey"

$p_3$ : LUGAR = "Puebla"

Los predicados simples para la segunda consulta serían:

$p_4$ : PRESUPUESTO £ 200000

$p_5$ : PRESUPUESTO > 200000

Si el algoritmo COM_MIN es seguido, el conjunto $Pr'$ = {$p_1$, $p_2$, $p_3$, $p_4$, $p_5$} es obviamente completo y mínimo. Basado en $Pr'$, los siguientes seis minitérminos que forman a $M$ se pueden definir como:

$m_1$: (LUGAR = "México") Ù (PRESUPUESTO £ 200000)

$m_2$: (LUGAR = "México") Ù (PRESUPUESTO > 200000)

$m_3$: (LUGAR = "Monterrey") Ù (PRESUPUESTO £ 200000)

$m_4$: (LUGAR = "Monterrey") Ù (PRESUPUESTO > 200000)

$m_5$: (LUGAR = "Puebla") Ù (PRESUPUESTO £ 200000)

$m_6$: (LUGAR = "Puebla") Ù (PRESUPUESTO > 200000)

Estos no son los únicos minitérminos que se pueden generar. Por ejemplo, es posible especificar predicados de la forma:

$p_1$ Ù $p_2$ Ù $p_3$ Ù $p_4$ Ù $p_5$

Sin embargo, las implicaciones obvias:

$i_1$: $p_1$ Þ Ø $p_2$ Ù Ø $p_3$

$i_2$: $p_2$ Þ Ø $p_1$ Ù Ø $p_3$

$i_3$: $p_3$ Þ Ø $p_1$ Ù Ø $p_2$

$i_4$: $p_4$ Þ Ø $p_5$

$i_5$: $p_5$ Þ Ø $p_4$

$i_6$: Ø $p_4$ Þ $p_5$

$i_7$: Ø $p_5$ Þ $p_4$

eliminan esos minitérminos y nos quedamos con $m_1$ hasta $m_6$. Observando la instancia de la base de datos del ejemplo, podríamos decir que las siguientes implicaciones se mantienen:

$i_1$: (LUGAR = "México") Ù NOT (PRESUPUESTO > 200000)

$i_2$: (LUGAR = "Monterrey") Ù NOT (PRESUPUESTO £ 200000)

$i_3$: NOT (LUGAR = "México") Ù (PRESUPUESTO £ 200000)

$i_4$: NOT (LUGAR = "Monterrey") Ù (PRESUPUESTO > 200000)

Sin embargo, recuerde que las implicaciones deben ser definidas de acuerdo a la semántica de la base de datos, no de acuerdo a los valores actuales. Algunos de los fragmentos definidos por

$M$ = { $m_1$, $m_2$, $m_3$, $m_4$, $m_5$, $m_6$ } pueden estar vacíos, pero ellos son, no obstante, fragmentos. No existe nada en la semántica de la base de datos que sugiera que las implicaciones $i_8$ hasta $i_{11}$ se satisfagan.

Los resultados de la fragmentación horizontal primaria de J forman seis fragmentos $F_J$ = { $J_1$, $J_2$, $J_3$, $J_4$, $J_5$, $J_6$ } de acuerdo a los minitérminos de *M.* Algunos de esos están vacíos y por lo tanto no se presentan aquí.

$J_1$

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J1 | Instrumentación | 150000 | Monterrey |

$J_3$

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J2 | Desarrollo de bases de datos | 135000 | México |

$J_4$

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J5 | CAD/CAM | 250000 | México |

$J_6$

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| J4 | Mantenimiento | 310000 | México |

..

**Correctitud de la Fragmentación Horizontal Primaria**

- **Completitud**. Ya que Pr' es completo y mínimo, los predicados de selección son completos.
- **Reconstrucción**. Si la relación $R$ es fragmentada en $F_R = (R_1, R_2, ..., R_r)$, entonces,

$$R = \text{U} \; "_{R_i \hat{\imath} F_R} R_i$$

- **Fragmentos disjuntos**. Los predicados minitérminos que forman la base de la fragmentación deben ser mutuamente exclusivos.

### 3.6 Fragmentación horizontal derivada

Una fragmentación horizontal derivada se define en la relación miembro de una liga de acuerdo a la operación de selección especificada en la relación propietaria. La liga entre las relaciones propietaria y miembro se define como una equi-junta. Una equi-junta se puede implementar por semi-juntas. Esto es importante, ya que se quiere particionar una relación miembro de acuerdo a la fragmentación de su

propietario, pero se quiere que los fragmentos resultantes queden definidos únicamente en los atributos de la relación miembro.

Dada una liga $L$ donde *owner*( $L$ ) = $S$ y *member*( $L$ ) = $R$, las ***fragmentos horizontal derivados*** de $R$ se definen como:

$R_i = R > p_F S_i$, 1 £ $i$ £ $w$

donde $w$ es el número máximo de fragmentos que serán definidos en $R$ y

$S_i = s_{Fi} ( S )$

donde $F_i$ es la fórmula de acuerdo a la cual la fragmentación horizontal primaria $S_i$ se define.

**Ejemplo 3.12**. Dada la liga L1 donde owner( $L_1$ ) = S y member( $L_1$ ) = E. Se pueden agrupar a los ingenieros en dos grupos con base en su salario; aquellos que perciben menos de 30000 y aquellos que ganan mas de 30000. Los fragmentos $E_1$ y $E_2$ se definen como sigue

$E_1 = E > p\ S_1$

$E_2 = E > p\ S_2$

donde

$S_1 = s_{SAL £ 30000} (S)$

$S_2 = s_{SAL > 30000} (S)$

Así, el resultado de la fragmentación se presenta en las siguientes tablas

$E_1$

| ENO | ENOMBRE | TITULO |
|---|---|---|
| E3 | Armando Legarreta | Ingeniero Mecánico |
| E4 | Beatriz Molleda | Programador |
| E7 | Roberto Dávila | Ingeniero Mecánico |

$E_2$

| ENO | ENOMBRE | TITULO |
|---|---|---|
| E1 | Juan Rodríguez | Ingeniero Eléctrico |
| E2 | Miguel Sánchez | Analista de Sistemas |
| E5 | Jorge Castañeda | Analista de Sistemas |
| E6 | Luis Chávez | Ingeniero Eléctrico |

| E8 | Julia Jiménez | Analista de Sistemas |
|---|---|---|

¨

Para llevar a cabo una fragmentación horizontal derivada se requieren tres entradas: el conjunto de particiones de la relación propietaria, la relación miembro, y el conjunto de predicados semi-junta entre el propietario y el miembro. El algoritmo de fragmentación es trivial y no será presentado aquí.

**Correctitud de la Fragmentación Horizontal Derivada**

- **Completitud**. La completitud de una fragmentación horizontal primaria se basa en la los predicados de selección usados. Siempre que los predicados de selección sean completos, la fragmentación resultante es completa. Ya que la base del algoritmo de fragmentación es el conjunto de predicados completos y mínimos, *Pr'*, la completitud se garantiza siempre y cuando no se cometan errores al definir *Pr'*.

  La completitud de una fragmentación horizontal derivada es un poco más difícil de definir. La dificultad se debe al hecho de que los predicados que determinan la fragmentación involucran a dos relaciones. Sea $R$ la relación miembro de una liga cuyo propietario es la relación $S$, la cual es fragmentada por $F_S = \{ S_1, S_2, ..., S_w \}$. Además, sea $A$ el atributo de junta entre $R$ y $S$. Entonces, por cada tupo $t$ en $R$, debe haber un tuplo $t'$ en $S$ tal que

  $$t[A] = t'[A]$$

  A esta regla se le conoce como integridad referencial y asegura que los tuplos de cualquier fragmento de la relación miembro están también en la relación propietaria.

- **Reconstrucción**. Si la relación $R$ es fragmentada en $F_R = (R_1, R_2, ..., R_w)$, entonces,

  $$R = \cup R_i, \forall R_i \in F_R$$

- **Fragmentos Disjuntos**. Es fácil establecer la condición de fragmentos disjuntos para fragmentación primaria. Sin embargo, para fragmentación derivada existe una semi-junta la cual incorpora cierta complejidad. Esta condición se puede garantizar si la gráfica de junta es simple. Si no es simple, es necesario consultar los valores de tuplos actuales. En general, no se quiere que un tuplo de una relación miembro se junte con dos o más tuplos de la relación propietario cuando esos tuplos están en diferentes fragmentos del propietario. Esto no siempre es fácil de establecer e illustra el porqué se desea siempre que los esquemas de fragmentación derivada tengan una gráfica de junta simple.

**Ejemplo 3.13**. En el ejemplo anterior los minitérminos predicados son

$$m_1: (SAL \leq 30000)$$

$$m_2: (SAL > 30000)$$

Ya que $m_1$ y $m_2$ son mutuamente exclusivos, la fragmentación de S es disjunta. Para la relación E, sin embargo, se requiere que

1. cada ingeniero tenga un solo título
2. cada título tenga un solo valor de salario asociado con él

Puesto que esas dos reglas se siguen de la semántica de la base de datos, la fragmentación de E con respecto a S es disjunta.

..

## 3.7 FRAGMENTACION VERTICAL

Una fragmentación vertical de una relación $R$ produce fragmentos $R_1$, $R_2$, ..., $R_r$, cada uno de los cuales contiene un subconjunto de los atributos de $R$ así como la llave primaria de $R$. El objetivo de la fragmentación vertical es particionar una relación en un conjunto de relaciones más pequeñas de manera que varias de las aplicaciones de usuario se ejecutarán sobre un fragmento. En este contexto, una fragmentación "óptima" es aquella que produce un esquema de fragmentación que minimiza el tiempo de ejecución de las consultas de usuario.

La fragmentación vertical ha sido estudiada principalmente dentro del contexto de los sistemas de manejo de bases de datos centralizados como una herramienta de diseño, la cual permite que las consultas de usuario traten con relaciones más pequeñas haciendo, por tanto, un número menor de accesos a páginas.

La fragmentación vertical es inherentemente más complicada que particionamiento horizontal ya que existe un gran número de alternativas para realizarla. Por lo tanto, se utilizan heurísticas para hacer el particionamiento. Los dos enfoques básicos son:

1. ***Agrupamiento***. Inicia asignando cada atributo a un fragmento, y en cada paso, algunos de los fragmentos satisfaciendo algún criterio se unen para formar un solo fragmento.
2. ***División***. Inicia con una sola relación realizar un particionamiento basado en el comportamiento de acceso de las consultas sobre los atributos.

Nos concentraremos aquí al estudio del enfoque divisional ya que, por un lado, su aplicación es más natural al enfoque de diseño "top-down". Además, el enfoque divisional genera fragmentos que no se traslapan mientras que el agrupamiento típicamente resulta en fragmentos traslapados. Por supuesto, la no traslapación no incluye a las llaves primarias.

### Requerimientos de información para la fragmentación vertical

Como en el caso de la fragmentación horizontal, es necesario proporcionar información para poder realizar una adecuada fragmentación vertical. Ya que el particionamiento vertical coloca en un fragmento aquellos atributos que se accesan juntos, se presenta la necesidad de una medida que relacione la afinidad de los atributos, la cual indica qué tan relacionados están los atributos. Esta medida se obtiene por datos primitivos.

Dado un conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ que serán aplicadas a la relación $R[A_1, A_2, ..., A_n]$, se define la función

$$use(q_i, A_j) = \begin{cases} 1 & \text{Si el atributo } A_j \text{ es referido por la consulta } q_i \\ 0 & \text{en caso contrario} \end{cases}$$

Los vectores $use(\ q_i,\ \cdot\ )$ son fáciles de definir si el diseñador conoce las aplicaciones que serán ejecutadas en la base de datos.

**Ejemplo 3.14**. Considere la relación J de la Figura 3.4. Suponga que las siguientes consultas se definen sobre esta relación:

$q_1$: Encuentre el presupuesto de un proyecto dado su número de identificación.

```
SELECT PRESUPUESTO
FROM J
WHERE JNO=valor
```

$q_2$: Encuentre los nombres y presupuestos de todos los proyectos.

```
SELECT JNOMBRE, PRESUPUESTO
FROM J
```

$q_3$: Encuentre los nombres de los proyectos en una ciudad dada.

```
SELECT JNOMBRE
FROM J
WHERE LUGAR=valor
```

$q_4$: Encuentre el presupuesto total de los proyectos en cada ciudad.

```
SELECT SUM(PRESUPUESTO)
FROM J
WHERE LUGAR=valor
```

Sean $A_1$=JNO, $A_2$=JNOMBRE, $A_3$=PRESUPUESTO, $A_4$=LUGAR. La función *use* se puede representar por la siguiente matriz:

$$\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

..

La medida de afinidad entre dos atributos $A_i$ y $A_j$ de una relación $R[A_1, A_2, ..., A_n]$ con respecto al conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ se define como sigue:

$$aff(A_i,A_j) = \text{S}_{\text{las consultas que accesan } A_i \text{ y } A_j} \text{ S}_{\text{Sl}} \ (ref_l(q_k)\ acc_l(q_k))$$

donde, $ref_l(q_k)$ es el número de accesos a los atributos $(A_i, A_j)$ para cada ejecución de la consulta $q_k$ en el sitio $S_l$ y $acc_l(q_k)$ es la frecuencia de acceso de la consulta previamente definida y modificada para incluir las frecuencias en sitios diferentes.

**Ejemplo 3.15**. Continuando con el ejemplo 3.14, suponga que cada consulta en dicho ejemplo accesa los atributos una vez durante cada ejecución ($ref_l(q_k) = 1$):

Las frecuencias de acceso de las consultas están dadas por:

$$\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array}\begin{array}{c} \begin{array}{ccc} S_1 & S_2 & S_3 \end{array} \\ \begin{bmatrix} 15 & 20 & 10 \\ 5 & 0 & 0 \\ 25 & 25 & 25 \\ 3 & 0 & 0 \end{bmatrix} \end{array}$$

La afinidad de los atributos A1 y A3 se puede medir como

$$aff(A_1, A_3) = \sum_{k=1}^{1}\sum_{l=1}^{3} acc_l(q_k) = acc_1(q_1) + acc_2(q_1) + acc_3(q_1) = 45$$

ya que la única aplicación que accesa ambos atributos es $q_1$. La matriz de afinidades entre atributos, *AA*, es

$$\begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array}\begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array}$$

..

**Algoritmo de Agrupamiento (Clustering)**

La tarea fundamental en el diseño de una fragmentación vertical es encontrar algún medio para agrupar los atributos de una relación basándose en los valores de afinidad entre atributos. La idea del algoritmo de agrupamiento es tomar la matriz de afinidades entre atributos (AA) y reorganizar el orden de los atributos para formar grupos en donde los atributos dentro de cada grupo presentan alta afinidad uno con otro.

El **algoritmo de energía acotada** (BEA por sus siglas en inglés) encuentra un ordenamiento de los atributos, de tal manera, que se maximiza la siguiente *medida de afinidad global* (AM):

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(A_i, A_j)\left[aff(A_i, A_{j-1}) + aff(A_i, A_{j+1}) + aff(A_{i-1}, A_j) + aff(A_i, A_{j-1})\right]$$

donde,

$$aff(A_0, A_j) = aff(A_i, A_0) = aff(A_{n+1}, A_j) = aff(A_i, A_{n+1}) = 0$$

**Algoritmo 3.3** BEA

Entrada: La matriz de afinidades entre atributos *AA*.

Salida: La matriz de afinidades agrupada, *CA*, la cual es una perturbación de *AA*.

*Iniciación:* Coloque y fije una de las columnas de *AA* en *CA*.

*Iteración:* Coloque las restantes *n-i* columnas en las restantes *i*+1 posiciones en la matriz *CA*. Para cada columna, elija la ubicación que causa la mayor contribución a la medida de afinidad global.

*Ordenamiento de renglones:* Ordene los renglones de acuerdo al ordenamiento de columnas.

Para definir la mejor ubicación se define la contribución de una ubicación.

$$cont(A_i, A_k, A_j) = 2bond(A_i, A_k) + 2bond(A_k, A_l) - 2bond(A_i, A_j)$$

donde,

$$bond(A_x, A_y) = \sum_{z=1}^{n} aff(A_z, A_x) aff(A_z, A_y)$$

**Ejemplo 3.16**. Considere la siguiente matriz *AA* y la matriz correspondiente *CA* en donde $A_1$ y $A_2$ han sido colocados.

$$AA = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & A_3 & A_4 \end{array} \\ \begin{bmatrix} 45 & 0 & 45 & 0 \\ 0 & 80 & 5 & 75 \\ 45 & 5 & 53 & 3 \\ 0 & 75 & 3 & 78 \end{bmatrix} \end{array} \quad CA = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & & \end{array} \\ \begin{bmatrix} 45 & 0 & & \\ 0 & 80 & & \\ 45 & 5 & & \\ 0 & 75 & & \end{bmatrix} \end{array}$$

$A_l$ colocar $A_3$ existen tres posibilidades:

- Ordenamiento(0-3-1):

$$cont(A_0, A_3, A_1) = 2bond(A_0, A_3) + 2bond(A_3, A_1) - 2bond(A_0, A_1)$$
$$= 2 * 0 + 2 * 4410 - 2 * 0 = 8820$$

- Ordenamiento(1-3-2):

$$cont(A_1, A_3, A_2) = 2bond(A_1, A_3) + 2bond(A_3, A_2) - 2bond(A_1, A_2)$$
$$= 2 * 4410 + 2 * 890 - 2 * 225 = 10150$$

- Ordenamiento(2-3-4):

$$cont(A_2, A_3, A_4) = 2bond(A_2, A_3) + 2bond(A_3, A_4) - 2bond(A_2, A_4)$$
$$= 1780$$

Por lo tanto, la matriz *CA* tiene la forma:

$$CA = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{ccc} A_1 & A_3 & A_2 \end{array} \\ \left[ \begin{array}{ccc} 45 & 45 & 0 \\ 0 & 5 & 80 \\ 45 & 53 & 5 \\ 0 & 3 & 75 \end{array} \quad \right] \end{array}$$

Cuando *A*4 es colocado, se obtiene la forma final de la matriz *CA* (después de la reorganización entre renglones):

$$CA = \begin{array}{c} \\ A_1 \\ A_3 \\ A_2 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_3 & A_2 & A_4 \end{array} \\ \left[ \begin{array}{cccc} 45 & 45 & 0 & 0 \\ 45 & 53 & 5 & 3 \\ 0 & 5 & 80 & 75 \\ 0 & 3 & 75 & 78 \end{array} \right] \end{array}$$

**Algoritmo de Particionamiento**

El objetivo del particionamiento es encontrar conjuntos de atributos que son accesados de manera única, o a lo más, por conjuntos disjuntos de consultas. Considere la matriz de atributos agrupada de la Figura 3.5. Si se fija un punto a lo largo de la diagonal, se identifican dos conjuntos de atributos. Un conjunto es $\{A_1, \ldots, A_i\}$ está en la esquina superior izquierda y el segundo conjunto $\{A_{i+1}, \ldots, A_n\}$ está en la esquina inferior derecha. Al primer conjunto se le llama *arriba* y al segundo conjunto se le denomina *abajo*.

Considere ahora el conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ y defina el conjunto de aplicaciones que accesan únicamente a *TA*, a *BA*, o ambas. Defina

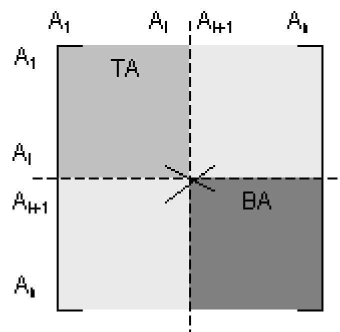



Figura 3.5. Localización del punto de división.

Considere ahora el conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ y defina el conjunto de aplicaciones que accesan únicamente a *TA*, a *BA*, o ambas. Defina

*TQ* = conjunto de aplicaciones que accesan únicamente a *TA*

*BQ* = conjunto de aplicaciones que accesan únicamente a *BA*

*OQ* = conjunto de aplicaciones que accesan tanto a *TA* como a *BA*

*CTQ* = número total de accesos a atributos por aplicaciones que accesan únicamente a *TA*

*CBQ* = número total de accesos a atributos por aplicaciones que accesan únicamente a *BA*

*COQ* = número total de accesos a atributos por aplicaciones que accesan únicamente tanto a *TA* como a *BA*

El problema es encontrar el punto a lo largo de la diagonal que maximiza la función objetivo

$$z = CTQ * CBQ - COQ^2$$

La característica importante de esta expresión es que define dos fragmentos tales que los valores de CTQ y CBQ son tan similares como sea posible. Esto nos permite balancear las cargas de procesamiento cuando los fragmentos están distribuidos en varios sitios.

Existen dos complicaciones que tienen que ser consideradas:

1. El particionamiento puede ser formado en la parte media de la matriz CA. Aquí se debe aplicar un corrimiento circular de un renglón hacia arriba y una columna hacia la izquierda para encontrar el mejor punto de particionamiento. Si esto se realiza para todos los posibles corrimientos el algoritmo tomaría $O(n^2)$ pasos.
2. Es posible que se formen más de dos grupos. Aquí la estrategia sería tratan con 1, 2, …, $n$-1 puntos a lo largo de la diagonal y tratar de hallar el mejor punto de particionamiento para cada uno de ellos. Claramente, este algoritmo tomaría entonces $O(2^n)$ pasos.

**Ejemplo 3.17**. Cuando el algoritmo de particionamiento se aplica a la matriz *CA* para la relación J, el resultado es la definición de los fragmentos $F_J = \{ J_1, J_2 \}$, donde $J_1 = \{A_1, A_3\}$ y $J_2 = \{A_1, A_2, A_4\}$. Así

$J_1$ = { JNO, PRESUPUESTO }

$J_2$ = { JNO, JNOMBRE, LUGAR }

..

**Correctitud de la Fragmentación Vertical**

- **Completitud**. La completitud de una fragmentación vertical es garantizada por el algoritmo de particionamiento. Ya que cada atributo de la relación global se asigna a uno de los fragmentos. Siempre y cuando el conjunto de atributos *A* sobre los cuales se define una relación *R* consiste de

  A = TA U TB

  la completitud de la fragmentación vertical se asegura.

- **Reconstrucción**. La reconstrucción de la relación global original se hace por medio de la operación de junta. Así, para una relación $R$ con fragmentación vertical $F_R = \{ R_1, R_2, ..., R_r \}$ y llave $K$

$$R = \bowtie_K R_i \; \forall R_i \in F_R$$

Por lo tanto, siempre que $R_i$ sea completo, la operación de junto reconstruirá adecuadamente $R$. Otro punto importante es que o cada $R_i$ debe contener a la llave de $R$, o debe contener los identificadores de tuplo asignados por el sistema (TID).

- **Fragmentos Disjuntos**. Existen dos casos:

1. Los TID no se considera que se traslapan ya que ellos son mantenidos por el sistema.
2. Las llaves duplicadas no se considera que se traslapan.

### 3.8 FRAGMENTACION HIBRIDA

En muchos casos una fragmentación horizontal o vertical de un esquema de una base de datos no será suficiente para satisfacer los requerimientos de aplicaciones de usuario. En este caso, una fragmentación vertical puede ser seguida de uno horizontal, o viceversa, produciendo un árbol de particionamiento estructurado, como se muestra en la Figura 3.6. Ya que los dos tipos de particionamiento se aplican uno después del otro, esta alternativa se le conoce como fragmentación ***híbrida***.

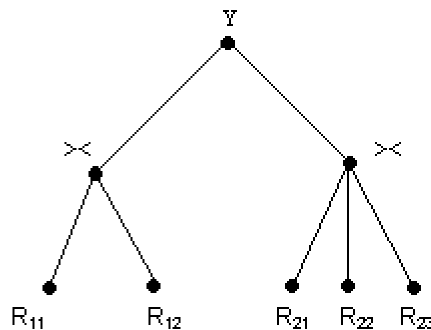



Figura 3.6. Fragmentación híbrida.

Un buen ejemplo de la necesidad de la fragmentación híbrida es la relación J, con la cual se ha trabajado. En la Figura 3.7 se muestra el árbol de reconstrucción de la fragmentación híbrida de J. Inicialmente se aplica una fragmentación horizontal y posteriormente una fragmentación vertical.

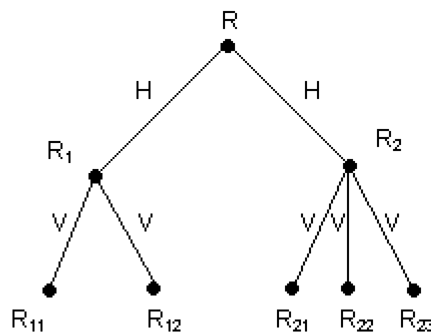



Figura 3.7. Fragmentación híbrida de la relación J.

## 3.9 ASIGNAMIENTO DE FRAGMENTOS

El asignamiento de recursos entre los nodos de una red de computadoras es un problema que se ha estudiado de manera extensa. Sin embargo, la mayoría de este trabajo no considera el problema de diseño de bases de datos distribuidas, en lugar de eso considera el problema de ubicar archivos individuales en redes de computadoras.

### El problema de asignamiento

Suponga que hay un conjunto de fragmentos $F = \{ F_1, F_2, ..., F_n \}$ y una red que consiste de los sitios $S = \{ S_1, S_2, ..., S_m \}$ en los cuales un conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ se van a ejecutar. El problema de asignamiento determina la distribución "óptima" de $F$ en $S$. La optimalidad puede ser definida de acuerdo a dos medidas:

1. *Costo mínimo*. Consiste del costo de comunicación de datos, del costo de almacenamiento, y del costo procesamiento (lecturas y actualizaciones a cada fragmento). El problema de asignamiento, entonces, pretende encontrar un esquema de asignmiento que minimiza una función de costo combinada.
2. *Rendimiento*. La estrategia de asignamiento se diseña para mantener una métrica de rendimiento. Las dos métricas más utilizadas son el tiempo de respuesta y el "throughput" (número de trabajos procesados por unidad de tiempo).

En cualquier problema de optimización existen restricciones que se deben satisfacer. El caso de distribución de fragmentos, las restricciones se establecen sobre las capacidades de almacenamiento y procesamiento de cada nodo en la red.

### Requerimientos de información

En la fase de asignamiento se necesita conocer información cuantitativa relativa a la base de datos, las aplicaciones que se utilizarán, la red de comunicaciones, las capacidades de procesamiento y de almacenamiento de cada nodo en la red.

- Información sobre la base de datos. Es necesario conocer la selectividad de un fragmento $F_j$ con respecto a una consulta $q_i$, esto es, el número de tuplos de $F_j$ que será necesario accesar para procesar $q_i$. Este valor se denota como $sel( F_j )$. Así también, es necesario conocer el tamaño de cada fragmento, el cual está dado por:

  $$size(F_j) = card(F_j) * length(F_j)$$

- Información sobre las aplicaciones. Es necesario distinguir el número de lecturas que una consulta $q_j$ hace a un fragmento $F_j$ durante su ejecución, del número de escrituras. Se requiere de una matriz que indique que consultas actualizan cuales fragmentos. Una matriz similar se necesita para indicar las lecturas de consultas a fragmentos. Finalmente, se necesita saber cual es el nodo de la red que origina cada consulta.

- Información sobre cada nodo de la red. Las medidas utilizadas son el costo unitario de almacenamiento de datos en un nodo y el costo unitario de procesamiento de datos en un nodo.
- Información sobre la red de comunicaciones. Las medidas a considerar son: la velocidad de comunicación, el tiempo de latencia en la comunicación y la cantidad de trabajo adicional a realizar para una comunicación.

**Asignamiento de archivos vs. Asignamiento de fragmentos**

En el diseño de bases de datos distribuidas no se puede considerar similar al problema de distribución de archivos por las siguientes razones:

1. Los fragmentos no son archivos individuales. La colocación de un fragmento usualmente tiene un impacto en la colocación de otros fragmentos. Por lo tanto, es necesario mantener las relaciones entre fragmentos.
2. El acceso a las bases de datos es más complicado que a archivos. Los modelos de acceso remoto a archivos no se aplican. Es necesario considerar las relaciones entre el asignamiento de fragmentos y el procesamiento de consultas.
3. El costo que incurre el mantenimiento de la integridad de la información debe ser considerado en las bases de datos distribuidas.
4. El costo que incurre el control de concurrencia a una base de datos distribuida también debe ser considerado.

**Modelo de Asignamiento**

Se discute ahora un modelo de asignamiento que pretende minimizar el costo total de procesamiento y almacenamiento satisfaciendo algunas restricciones en el tiempo de respuesta. El modelo tiene la siguiente forma general:

min( Costo Total )

dadas

restricciones en el tiempo de respuesta

restricciones en las capacidades de almacenamiento

restricciones en el tiempo de procesamiento

A continuación se tratará de ampliar las componentes de este modelo. Se define la variable de decisión $x_{ij}$ de la siguiente manera:

$$x_{ij} = \begin{cases} 1 & \text{si el fragmento } F_i \text{ es almacenado en el nodo } S_j \\ 0 & \text{en otro caso} \end{cases}$$

**Costo total**

La función de costo total tiene dos componentes: procesamiento de consultas y almacenamiento. Así, puede ser expresado de la siguiente forma:

$$TOC = \sum_{\forall q_i \in Q} QPC_i + \sum_{\forall s_k \in S} \sum_{\forall F_j \in F} STC_{jk}$$

donde $QPC_i$ es el costo de procesamiento de la consulta $q_i$, y $STC_{jk}$ es el costo de almacenar el fragmento $F_j$ en el nodo $S_k$.

El costo de almacenamiento se puede expresar como

$$STC_{jk} = USC_k * size( F_j ) * x_{jk}$$

donde $USC_k$ es el costo de almacenamiento unitario en el nodo $S_k$.

El costo de procesamiento de una consulta tiene dos componentes: el costo de procesamiento y el costo de transmisión. Esto se puede expresar como:

$$QPC_i = PC_i + TC_i$$

La componente de procesamiento involucra tres factores: el costo acceso (*AC*), el costo de mantenimiento de la integridad (*IE*) y el costo debido al control de concurrencia (*CC*). Así podemos expresar:

$$PC_i = AC_i + IE_i + CC_i$$

La especificación detallada de cada uno de esos factores de costo depende del algoritmo utilizado para realizar estas tareas. Sin embargo, el costo de acceso se puede especificar con algún detalle:

$$AC_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} (u_{ij} * UR_{ij} + r_{ij} * RR_{ij}) * x_{jk} * LPC_k$$

donde los primeros dos términos dan el número total de actualizaciones y lecturas realizadas por la consulta $q_i$ en el fragmento $F_j$, y $LPC_k$ es el costo unitario de procesamiento local, en $S_k$, de una unidad de trabajo.

Los costos del mantenimiento de la integridad y del control de concurrencia pueden ser calculados similarmente al costo de acceso. Sin embargo, éstos no se discutirán sino en los capítulos siguientes.

Respecto a la componente de transmisión, ésta puede separarse en el procesamiento de actualizaciones y de consultas (lecturas), dado que los tiempos de procesamiento para ellas son completamente diferentes. En las actualizaciones, es necesario informar a todos los nodos con réplicas, mientras que en las lecturas o consultas, es suficiente con accesar solo una de las copias. Más aún, al final de una solicitud de actualización, no existe una transmisión de datos de regreso mas que un mensaje de confirmación, mientras que una consulta puede resultar una transmisión significativa de datos.

La componente de actualizaciones de la función de transmisión es

$$TCU_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{o(i),k} + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{k,o(i)}$$

El primer término es por el envío del mensaje de actualización desde el nodo de origen $o(i)$ de $q_i$ a todos los fragmentos con réplicas que necesitan ser actualizados. El segundo término es debido al mensaje confirmación. El costo de consulta se puede especificar como:

$$TCR_i = \sum_{\forall F_j \in F} \min_{S_k \in S} \left( u_{ij} * x_{ij} * g_{o(i),k} + r_{ij} * x_{ij} * \frac{sel_i(F_j)}{fsize} * g_{k,o(i)} \right)$$

El primer término en TCR representa el costo de transmitir la solicitud de consulta a aquellos nodos que contienen copias de los fragmentos que necesitan ser accesados. El segundo término toma en cuenta la transmisión de los resultados de esos nodos al nodo de origen. La ecuación sólo considera de entre los nodos con copias del mismo fragmento, solo el nodo que produce el costo mínimo de transmisión. Ahora, la función del costo de transmisión para la consulta $q_i$ puede ser especificada como:

$$TC_i = TCU_i + TCR_i$$

**Restricciones**

Las funciones de restricción se pueden especificar con un detalle similar a la función de costo total. Sin embargo, en lugar de describir tales funciones con profundidad, se indicará simplemente cual es su forma general. La restricción del tiempo de respuesta se debe especificar como:

tiempo de ejecución de $q_i$ £ máximo tiempo de respuesta de $q_i$, " $q_i$ Î $Q$

La restricción de almacenamiento se puede especificar como:

$$\sum STC_{jk} \leq \text{capacidad de almacenamiento en el nodo } q_i, \ \forall q_i \in Q$$

La restricción del tiempo de procesamiento es:

$$\sum_{\forall q_i \in Q} \text{carga de procamiento de } q_i \text{ en el nodo } S_j \leq \text{capacidad de procesamiento de } S_k, \ \forall S_k \in S$$

**Métodos de solución**

Es sabido que el problema de asignamiento establecido como en el modelo discutido pertenece a la clase de problemas NP-completos. Por lo tanto, es necesario buscar métodos heurísticos que produzcan soluciones aproximadas. Diferentes heurísticas se han usado a la solución del modelo de asignamiento entre las cuales se pueden mencionar: la solución al problema de la valija (knapsack), técnicas tipo "branch-and-bound" y algoritmos para el flujo de redes.

Ha habido varios intentos para reducir la complejidad del problema. Una estrategia ha sido asumir que todos los particionamientos posibles han sido determinados junto con sus costos asociados y sus beneficios en términos del procesamiento de consultas. El problema entonces, es modelado como la elección del particionamiento y asignamiento óptimos para cada relación. Otra simplificación frecuentemente empleada es ignorar inicialmente la replicación de datos y enconcontrar una solución óptima para el caso no replicado. La replicación se incorpora en un segundo paso el cual aplica un algoritmo ávido que inicia a partir de la solución no replicada y trata de mejorarla iterativamente.

3.9 ASIGNAMIENTO DE FRAGMENTOS

El asignamiento de recursos entre los nodos de una red de computadoras es un problema que se ha estudiado de manera extensa. Sin embargo, la mayoría de este trabajo no considera el problema de

diseño de bases de datos distribuidas, en lugar de eso considera el problema de ubicar archivos individuales en redes de computadoras.

**El problema de asignamiento**

Suponga que hay un conjunto de fragmentos $F = \{ F_1, F_2, ..., F_n \}$ y una red que consiste de los sitios $S = \{ S_1, S_2, ..., S_m \}$ en los cuales un conjunto de consultas $Q = \{ q_1, q_2, ..., q_q \}$ se van a ejecutar. El problema de asignamiento determina la distribución "óptima" de $F$ en $S$. La optimalidad puede ser definida de acuerdo a dos medidas:

1. *Costo mínimo*. Consiste del costo de comunicación de datos, del costo de almacenamiento, y del costo procesamiento (lecturas y actualizaciones a cada fragmento). El problema de asignamiento, entonces, pretende encontrar un esquema de asignmiento que minimiza una función de costo combinada.
2. *Rendimiento*. La estrategia de asignamiento se diseña para mantener una métrica de rendimiento. Las dos métricas más utilizadas son el tiempo de respuesta y el "throughput" (número de trabajos procesados por unidad de tiempo).

En cualquier problema de optimización existen restricciones que se deben satisfacer. El caso de distribución de fragmentos, las restricciones se establecen sobre las capacidades de almacenamiento y procesamiento de cada nodo en la red.

**Requerimientos de información**

En la fase de asignamiento se necesita conocer información cuantitativa relativa a la base de datos, las aplicaciones que se utilizarán, la red de comunicaciones, las capacidades de procesamiento y de almacenamiento de cada nodo en la red.

- Información sobre la base de datos. Es necesario conocer la selectividad de un fragmento $F_j$ con respecto a una consulta $q_i$, esto es, el número de tuplos de $F_j$ que será necesario accesar para procesar $q_i$. Este valor se denota como $sel( F_j )$. Así también, es necesario conocer el tamaño de cada fragmento, el cual está dado por:

$$size(F_j) = card( F_j ) * length( F_j )$$

- Información sobre las aplicaciones. Es necesario distinguir el número de lecturas que una consulta $q_j$ hace a un fragmento $F_j$ durante su ejecución, del número de escrituras. Se requiere de una matriz que indique que consultas actualizan cuales fragmentos. Una matriz similar se necesita para indicar las lecturas de consultas a fragmentos. Finalmente, se necesita saber cual es el nodo de la red que origina cada consulta.
- Información sobre cada nodo de la red. Las medidas utilizadas son el costo unitario de almacenamiento de datos en un nodo y el costo unitario de procesamiento de datos en un nodo.
- Información sobre la red de comunicaciones. Las medidas a considerar son: la velocidad de comunicación, el tiempo de latencia en la comunicación y la cantidad de trabajo adicional a realizar para una comunicación.

**Asignamiento de archivos vs. Asignamiento de fragmentos**

En el diseño de bases de datos distribuidas no se puede considerar similar al problema de distribución de archivos por las siguientes razones:

1. Los fragmentos no son archivos individuales. La colocación de un fragmento usualmente tiene un impacto en la colocación de otros fragmentos. Por lo tanto, es necesario mantener las relaciones entre fragmentos.
2. El acceso a las bases de datos es más complicado que a archivos. Los modelos de acceso remoto a archivos no se aplican. Es necesario considerar las relaciones entre el asignamiento de fragmentos y el procesamiento de consultas.
3. El costo que incurre el mantenimiento de la integridad de la información debe ser considerado en las bases de datos distribuidas.
4. El costo que incurre el control de concurrencia a una base de datos distribuida también debe ser considerado.

**Modelo de Asignamiento**

Se discute ahora un modelo de asignamiento que pretende minimizar el costo total de procesamiento y almacenamiento satisfaciendo algunas restricciones en el tiempo de respuesta. El modelo tiene la siguiente forma general:

min( Costo Total )

dadas

restricciones en el tiempo de respuesta

restricciones en las capacidades de almacenamiento

restricciones en el tiempo de procesamiento

A continuación se tratará de ampliar las componentes de este modelo. Se define la variable de decisión $x_{ij}$ de la siguiente manera:

$$x_{ij} = \begin{cases} 1 & \text{si el fragmento } F_i \text{ es almacenado en el nodo } S_j \\ 0 & \text{en otro caso} \end{cases}$$

**Costo total**

La función de costo total tiene dos componentes: procesamiento de consultas y almacenamiento. Así, puede ser expresado de la siguiente forma:

$$TOC = \sum_{\forall q_i \in Q} QPC_i + \sum_{\forall s_k \in S} \sum_{\forall F_j \in F} STC_{jk}$$

donde $QPC_i$ es el costo de procesamiento de la consulta $q_i$, y $STC_{jk}$ es el costo de almacenar el fragmento $F_j$ en el nodo $S_k$.

El costo de almacenamiento se puede expresar como

$$STC_{jk} = USC_k * size(F_j) * x_{jk}$$

donde $USC_k$ es el costo de almacenamiento unitario en el nodo $S_k$.

El costo de procesamiento de una consulta tiene dos componentes: el costo de procesamiento y el costo de transmisión. Esto se puede expresar como:

$$QPC_i = PC_i + TC_i$$

La componente de procesamiento involucra tres factores: el costo acceso (*AC*), el costo de mantenimiento de la integridad (*IE*) y el costo debido al control de concurrencia (*CC*). Así podemos expresar:

$$PC_i = AC_i + IE_i + CC_i$$

La especificación detallada de cada uno de esos factores de costo depende del algoritmo utilizado para realizar estas tareas. Sin embargo, el costo de acceso se puede especificar con algún detalle:

$$AC_i = \sum_{\forall s_k \in S} \sum_{\forall F_j \in F} (u_{ij} * UR_{ij} + r_{ij} * RR_{ij}) * x_{jk} * LPC_k$$

donde los primeros dos términos dan el número total de actualizaciones y lecturas realizadas por la consulta $q_i$ en el fragmento $F_j$, y $LPC_k$ es el costo unitario de procesamiento local, en $S_k$, de una unidad de trabajo.

Los costos del mantenimiento de la integridad y del control de concurrencia pueden ser calculados similarmente al costo de acceso. Sin embargo, éstos no se discutirán sino en los capítulos siguientes.

Respecto a la componente de transmisión, ésta puede separarse en el procesamiento de actualizaciones y de consultas (lecturas), dado que los tiempos de procesamiento para ellas son completamente diferentes. En las actualizaciones, es necesario informar a todos los nodos con réplicas, mientras que en las lecturas o consultas, es suficiente con accesar solo una de las copias. Más aún, al final de una solicitud de actualización, no existe una transmisión de datos de regreso mas que un mensaje de confirmación, mientras que una consulta puede resultar una transmisión significativa de datos.

La componente de actualizaciones de la función de transmisión es

$$TCU_i = \sum_{\forall s_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{o(i),k} + \sum_{\forall s_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{k,o(i)}$$

El primer término es por el envío del mensaje de actualización desde el nodo de origen *o*(*i*) de $q_i$ a todos los fragmentos con réplicas que necesitan ser actualizados. El segundo término es debido al mensaje confirmación. El costo de consulta se puede especificar como:

$$TCR_i = \sum_{\forall F_j \in F} \min_{S_k \in S} \left( u_{ij} * x_{ij} * g_{o(i),k} + r_{ij} * x_{ij} * \frac{sel_i(F_j)}{fsize} * g_{k,o(i)} \right)$$

El primer término en TCR representa el costo de transmitir la solicitud de consulta a aquellos nodos que contienen copias de los fragmentos que necesitan ser accesados. El segundo término toma en cuenta la transmisión de los resultados de esos nodos al nodo de origen. La ecuación sólo considera de entre los nodos con copias del mismo fragmento, solo el nodo que produce el costo mínimo de transmisión. Ahora, la función del costo de transmisión para la consulta $q_i$ puede ser especificada como:

$TC_i = TCU_i + TCR_i$

### Restricciones

Las funciones de restricción se pueden especificar con un detalle similar a la función de costo total. Sin embargo, en lugar de describir tales funciones con profundidad, se indicará simplemente cual es su forma general. La restricción del tiempo de respuesta se debe especificar como:

$$\text{tiempo de ejecución de } q_i \leq \text{máximo tiempo de respuesta de } q_i,\ \forall q_i \in Q$$

La restricción de almacenamiento se puede especificar como:

$$\sum STC_{jk} \leq \text{capacidad de almacenamiento en el nodo } q_i,\ \forall q_i \in Q$$

La restricción del tiempo de procesamiento es:

$$\sum_{\forall q_i \in Q} \text{carga de procamiento de } q_i \text{ en el nodo } S_j \leq \text{capacidad de procesamiento de } S_k,\ \forall S_k \in S$$

### Métodos de solución

Es sabido que el problema de asignamiento establecido como en el modelo discutido pertenece a la clase de problemas NP-completos. Por lo tanto, es necesario buscar métodos heurísticos que produzcan soluciones aproximadas. Diferentes heurísticas se han usado a la solución del modelo de asignamiento entre las cuales se pueden mencionar: la solución al problema de la valija (knapsack), técnicas tipo "branch-and-bound" y algoritmos para el flujo de redes.

Ha habido varios intentos para reducir la complejidad del problema. Una estrategia ha sido asumir que todos los particionamientos posibles han sido determinados junto con sus costos asociados y sus beneficios en términos del procesamiento de consultas. El problema entonces, es modelado como la elección del particionamiento y asignamiento óptimos para cada relación. Otra simplificación frecuentemente empleada es ignorar inicialmente la replicación de datos y enconcontrar una solución óptima para el caso no replicado. La replicación se incorpora en un segundo paso el cual aplica un algoritmo ávido que inicia a partir de la solución no replicada y trata de mejorarla iterativamente.

# CAPITULO 4. PROCESAMIENTOS DE CONSULTAS DISTRIBUIDAS

El éxito creciente de la tecnología de bases de datos relacionales en el procesamiento de datos se debe, en parte, a la disponibilidad de lenguajes no procedurales los cuales pueden mejorar significativamente el desarrollo de aplicaciones y la productividad del usuario final. Ocultando los detalles de bajo nivel acerca de la localización física de datos, los lenguajes de bases de datos relacionales permiten la expresión de consultas complejas en una forma concisa y simple. Particularmente, para construir la respuesta a una consulta, el usuario no tiene que especificar de manera precisa el procedimiento que se debe seguir. Este procedimiento es llevado a cabo por un módulo del DBMS llamado el *procesador de consultas* (query processor).

Dado que la ejecución de consultas es un aspecto crítica en el rendimiento de un DBMS, el procesamiento de consultas ha recibido una gran atención tanto para bases de datos centralizadas como distribuidas. Sin embargo, el procesamiento de consultas es mucho más difícil en ambientes distribuidos que en centralizados, ya que existe un gran número de parámetros que afectan el rendimiento de las consultas distribuidas. En este capítulo revisaremos el procesamiento de consultas para bases de datos distribuidas.

### 4.1 El problema de procesamiento de consultas

La función principal de un procesador de consultas relacionales es transformar una consulta en una especificación de alto nivel, típicamente en cálculo relacional, a una consulta equivalente en una especificación de bajo nivel, típicamente alguna variación del álgebra relacional (ver Figura 4.1). La consulta de bajo nivel implementa de hecho la estrategia de ejecución para la consulta. La transformación debe ser correcta y eficiente. Es correcta si la consulta de bajo nivel tiene la misma semántica que la consulta original, esto es, si ambas consultas producen el mismo resultado. El mapeo bien definido que se conoce entre el cálculo relacional y el álgebra relacional hace que la correctitud de la transformación sea fácil de verificar. Sin embargo, producir una estrategia de ejecución eficiente es mucho más complicado. Una consulta en el cálculo relacional puede tener muchas transformaciones correctas y equivalentes en el álgebra relacional. Ya que cada estrategia de ejecución equivalente puede conducir a consumos de recursos de cómputo muy diferentes, la dificultad más importante es seleccionar la estrategia de ejecución que minimiza el consumo de recursos.

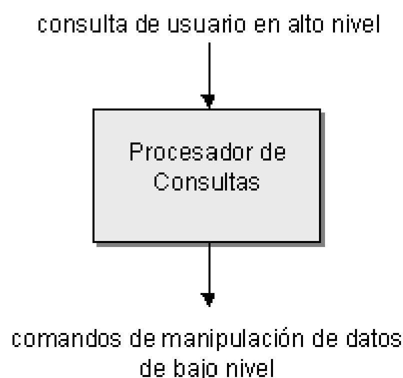



Figura 4.1. Procesamiento de consultas.

**Ejemplo 4.1**. Considere el siguiente subconjunto del esquema de la base de datos de ingeniería que se presentó en el capítulo 2

E( ENO, ENOMBRE, TITULO )

G( ENO, JNO, RESPONSABLE, JORNADA )

y la siguiente consulta de usuario:

"Encuentre todos los nombres de empleados que manejan un proyecto"

La expresión de la consulta en SQL se puede ver como

**SELECT** ENOMBRE

**FROM** E, G

**WHERE** E.ENO = G.ENO **AND** RESPONSABLE = "ADMINISTRADOR"

Dos consultas equivalentes en el álgebra relacional que son transformaciones correctas de la consulta en SQL son:

$$\prod_{ENAME}(\sigma_{RESP="ADMINSTRADOR" \wedge E.ENO=G.ENO}(E \times G))$$

y

$$\prod_{ENAME}(E \bowtie_{ENO} (\sigma_{RESP="ADMINSTRADOR"}(G))$$

Como es intuitivamente obvio, la segunda estrategia que evita calcular el producto cartesiano entre E y G, consume mucho menos recursos que la primera y, por lo tanto, es mejor.

..

En un contexto centralizado, las estrategias de ejecución de consultas pueden ser bien expresadas como una extensión del álgebra relacional. Sin embargo, en sistemas distribuidos, el álgebra relacional no es suficiente para expresar la ejecución de estrategias. Debe ser complementada con operaciones para el intercambio de datos entre nodos diferentes. Además de elegir el orden de las operaciones del álgebra relacional, el procesador de consultas distribuidas debe seleccionar también los mejores sitios para procesar datos y posiblemente la forma en que ellos tienen que ser transformados.

**Ejemplo 4.2**. Considere la siguiente consulta del Ejemplo 4.1:

$$\prod_{ENAME}(E \bowtie_{ENO} (\sigma_{RESP="ADMINSTRADOR"}(G))$$

Supongamos que las relaciones E y G están fragmentadas horizontalmente como sigue:

$E_1 = s_{ENO \pounds "E3"} (E)$

$E_2 = s_{ENO > "E3"} (E)$

$G_1 = s_{ENO \pounds "E3"} (G)$

$G_2 = s_{ENO > "E3"} (G)$

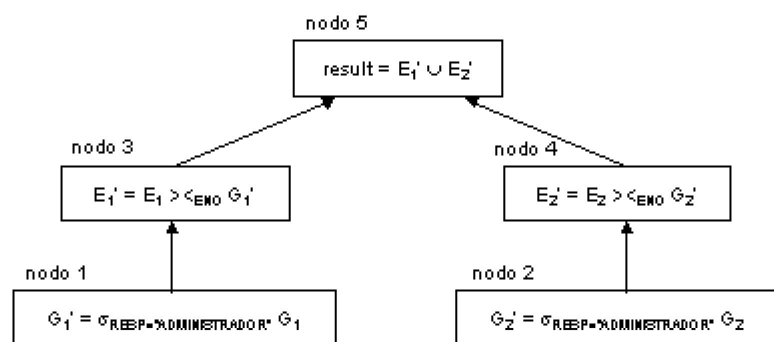
Los fragmentos $E_1$, $E_2$, $G_1$ y $G_2$ están almacenados en los nodos 1, 2, 3 y 4, respectivamente, y el resultado se quiere en el nodo 5. En la Figura 4.2 se presentan dos estrategias distribuidas de ejecución diferentes para la misma consulta (se ha ignorado el operador de proyección por simplicidad del ejemplo). La estrategia A explota el hecho de que las relaciones E y G están fragmentadas de la misma

manera y ejecuta la operación de selección y junta en paralelo. La estrategia B centraliza todos los datos en el nodo resultante antes de procesar la consulta.

Para evaluar el consumo de recursos, se usará un modelo de costo simple. Suponga que el costo de acceso a un tuplo (*tupacc*) es 1 unidad, y la transferencia de un tuplo (*tuptrans*) tiene un costo de 10 unidades. Suponga que las relaciones E y G tienen 400 y 1000 tuplos, respectivamente, y que existen 20 administradores en la relación G. También, suponga que los datos están uniformemente distribuidos entre los nodos. Finalmente, suponga que las relaciones G y E están agrupadas localmente en los atributos RESP y ENO, respectivamente, de manera que, hay un acceso directo a los tuplos de G y E, respectivamente.

El costo de la estrategia A se puede derivar como sigue:

| | | |
|---|---|---|
| 1. Producir G' seleccionando G requiere 20**tupacc* | = | 20 |
| 2. Transferir G' a los nodos de E requiere 20**tuptrans* | = | 200 |
| 3. Producir E' juntando G' y E requiere (10*10)**tupacc**2 | = | 200 |
| 4. Transferir E' al nodo resultante requiere 20**tuptrans* | = | <u>200</u> |
| El costo total es | | 620 |



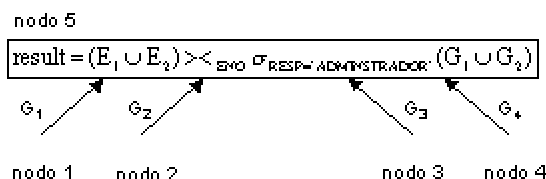




Figura 4.2. Estrategias de ejecución distribuida equivalentes.

El costo de la estrategia B se puede derivar como sigue:

| | | |
|---|---|---|
| 1. Transferir E al nodo 5 requiere 400**tuptrans* | = | 4,000 |
| 2. Transferir G al nodo 5 requiere 1000**tuptrans* | = | 10,000 |
| 3. Producir G' seleccionando G 1000**tupacc* | = | 1,000 |
| 4. Juntar E y G' requiere 1000*20**tupacc* | = | <u>20,000</u> |
| El costo total es | | 30,000 |

La estrategia A es mejor por un factor de 37, lo cual es significativo. La diferencia sería aún mayor si se hubiera asumido un costo de comunicación mayor y/o un grado de fragmentación mayor.

..

### 4.2 Objetivos de la optimización de consultas

Como se estableció antes, el objetivo del procesamiento de consultas en un ambiente distribuido es transformar una consulta sobre una base de datos distribuida en una especificación de alto nivel a una estrategia de ejecución eficiente expresada en un lenguaje de bajo nivel sobre bases de datos locales.

Así, el problema de optimización de consultas es minimizar una función de costo tal que

*función de costo total* = *costo de* I/O + *costo de* CPU + *costo de comunicación*

Los diferentes factores pueden tener pesos diferentes dependiendo del ambiente distribuido en el que se trabaje. Por ejemplo, en las redes de área amplia (WAN), normalmente el costo de comunicación domina dado que hay una velocidad de comunicación relativamente baja, los canales están saturados y el trabajo adicional requerido por los protocolos de comunicación es considerable. Así, los algoritmos diseñados para trabajar en una WAN, por lo general, ignoran los costos de CPU y de I/O. En redes de área local (LAN) el costo de comunicación no es tan dominante, así que se consideran los tres factores con pesos variables.

### 4.3 La complejidad de las operaciones del álgebra relacional

La complejidad de las operaciones del álgebra relacional afectan directamente su tiempo de ejecución y establecen algunos principios útiles al procesador de consultas. Esos principios pueden ayudar en elegir la estrategia de ejecución final. La forma más simple de definir la complejidad es en términos de la cardinalidad de las relaciones independientemente de los detalles de implementación tales como fragmentación y estructuras de almacenamiento. La Figura 4.3 presenta la complejidad de las operaciones unarias y binarias en el orden creciente de complejidad.

| **Operación** | **Complejidad** |
|---|---|
| Selección<br>Proyección<br>(sin eliminación de duplicados) | $O(n)$ |
| Proyección<br>(con eliminación de duplicados)<br>Agrupación | $O(n*\log n)$ |
| Junta<br>Semijunta<br>División<br>Operadores sobre conjuntos | $O(n*\log n)$ |
| Producto Cartesiano | $O(n^2)$ |

Figura 4.3. Complejidad de las operaciones del álgebra relacional.

Esta simple mirada a la complejidad de las operaciones sugiere dos principios:

1. Dado que la complejidad es con base en las cardinalidades de las relaciones, las operaciones más selectivas que reducen las cardinalidades deben ser ejecutadas primero.
2. Las operaciones deben ser ordenadas en el orden de complejidad creciente de manera que el producto Cartesiano puede ser evitado o, al menos, ejecutado al final de la estrategia.

### 4.4 Caracterización de los procesadores de consultas

Es difícil evaluar y comparar procesadores de consultas para sistemas centralizados y distribuidos dado que ellos difieren en muchos aspectos. En esta sección se enumeran algunas características importantes de los procesadores de consultas que pueden ser usados como base para su comparación.

**Tipo de optimización**

El problema de optimización de consultas es altamente demandante en tiempo de ejecución y, en el caso general, es un problema de la clase *NP*. Así existen dos estrategias para su solución: búsqueda exhaustiva o el uso de heurísticas. Los algoritmos de búsqueda exhaustiva tienen una complejidad combinatorial en el número de relaciones de la consulta. Obtienen la transformación óptima, pero sólo se aplican a consultas simples dado su tiempo de ejecución.

Por otro lado, los algoritmos heurísticos obtienen solo aproximaciones a la transformación óptima pero lo hacen en un tiempo de ejecución razonable. Las heurísticas más directas a aplicar son el agrupamiento de expresiones comunes para evitar el cálculo repetido de las mismas, aplicar primero las operaciones de selección y proyección, reemplazar una junta por una serie de semijuntas y reordenar operaciones para reducir el tamaño de las relaciones intermedias.

**Granularidad de la optimización**

Existen dos alternativas: considerar sólo una consulta a la vez o tratar de optimizar múltiples consultas. La primera alternativa no considera el uso de resultados comunes intermedios. En el segundo caso puede obtener transformaciones eficientes si las consultas son similares. Sin embargo, el espacio de decisión es mucho más amplio lo que afecta grandemente el tiempo de ejecución de la optimización.

**Tiempo de optimización**

Una consulta puede ser optimizada en tiempos diferentes con relación a tiempo de ejecución de la consulta. La optimización se puede realizar de manera *estática* antes de ejecutar la consulta o de forma *dinámica* durante la ejecución de la consulta. La optimización estática se hace en tiempo de compilación de la consulta. Así, el costo de la optimización puede ser amortizada sobre múltiples ejecuciones de la misma consulta.

Durante la optimización de consultas dinámica la elección de la mejor operación siguiente se puede hacer basado en el conocimiento exacto de los resultados de las operaciones anteriores. Por tanto, se requiere tener estadísticas acerca del tamaño de los resultados intermedios para aplicar esta estrategia.

Un tercer enfoque, conocido como *híbrido*, utiliza básicamente un enfoque estático, pero se puede aplicar un enfoque dinámico cuando los tamaños de las relaciones estimados están alejados de los tamaños actuales.

**Estadísticas**

La efectividad de una optimización recae en las *estadísticas* de la base de datos. La optimización dinámica de consultas requiere de estadísticas para elegir las operaciones que deben realizarse primero. La optimización estática es aún más demandante ya que el tamaño de las relaciones intermedias también debe ser estimado basándose en estadísticas. En bases de datos distribuidas las estadísticas para optimización de consultas típicamente se relacionan a los fragmentos; la cardinalidad y el tamaño de los fragmentos son importantes así como el número de valores diferentes de los atributos. Para minimizar la probabilidad de error, estadísticas más detalladas tales como histogramas de valores de atributos se usan pagando un costo mayor por su manejo.

**Nodos de Decisión**

Cuando se utiliza la optimización estática, un solo nodo o varios de ellos pueden participar en la selección de la estrategia a ser aplicada para ejecutar la consulta. La mayoría de los sistemas utilizan un enfoque centralizado para la toma de decisiones en el cual un solo lugar decide la estrategia a ejecutar. Sin embargo, el proceso de decisión puede ser distribuido entre varios nodos los cuales participan en la elaboración de la mejor estrategia. El enfoque centralizado es simple, pero requiere tener conocimiento de la base de datos distribuida completa. El enfoque distribuido requiere solo de información local. Existen enfoques híbridos en donde un nodo determina una calendarización global de las operaciones de la estrategia y cada nodo optimiza las subconsultas locales.

**Topología de la Red**

Como se mencionó al principio, el tipo de red puede impactar severamente la función objetivo a optimizar para elegir la estrategia de ejecución. Por ejemplo, en redes de tipo WAN se sabe que en la función de costo el factor debido a las comunicaciones es dominante. Por lo tanto, se trata de crear una calendarización global basada en el costo de comunicación. A partir de ahí, se generan calendarizaciones locales de acuerdo a una optimización de consultas centralizada. En redes de tipo LAN el costo de comunicación no es tan dominante. Sin embargo, se puede tomar ventaja de la comunicación "broadcast" que existe comúnmente en este tipo de redes para optimizar el procesamiento de las operaciones junta. Por otra parte, se han desarrollado algoritmos especiales para topologías específicas, como por ejemplo, la topología de estrella.

**4.5 Arquitectura del procesamiento de consultas**

El problema de procesamiento de consultas se puede descomponer en varios subproblems que corresponden a diferentes niveles. En la Figura 4.4, se presenta un esquema por niveles genérico para el procesamiento de consultas. Para simplificar la discusión, suponga que se tiene un procesador de consultas estático semicentralizado en donde no se tienen fragmentos replicados. Cuatro capas principales están involucradas en mapear una consulta a una base de datos distribuida en una secuencia optimizada de operaciones locales, cada una de ellas actuando en una base de datos local. Las cuatro capas principales son: *descomposición de consultas*, *localización de datos*, *optimización global de consultas* y *optimización local de consultas*. Las primeras tres se realizan en un nodo central usando información global. La cuarta capa se realiza en cada nodo local.

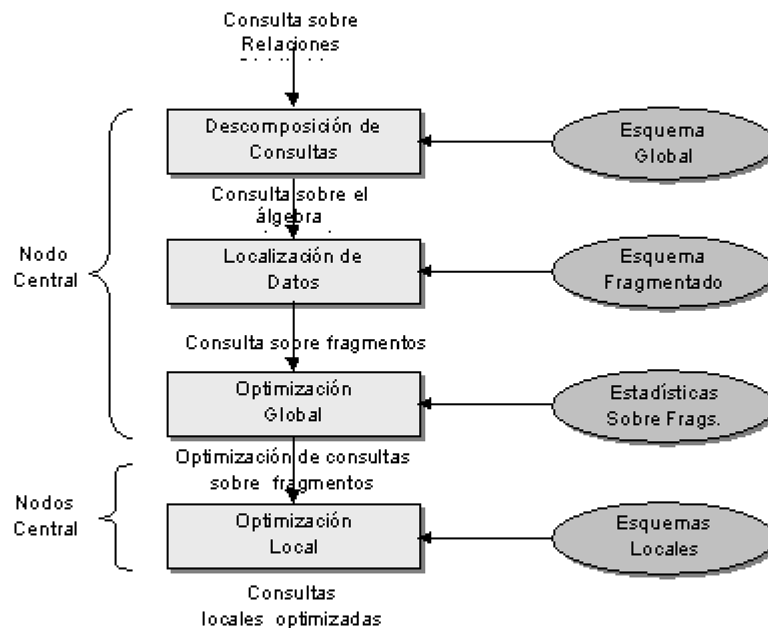


Figura 4.4. Arquitectura en capas del procesamiento de consultas.

**Descomposición de consultas**

La primera capa descompone una consulta en el cálculo relacional en una consulta en el álgebra relacional que opera sobre relaciones globales. Consiste de cuatro partes:

1. **Normalización**. Involucra la manipulación de los cuantificadores de la consulta y de los calificadores de la misma mediante la aplicación de la prioridad de los operadores lógicos.
2. **Análisis**. Se detecta y rechazan consultas semánticamente incorrectas.
3. **Simplificación**. Elimina predicados redundantes.
4. **Reestructuración**. Mediante reglas de transformación una consulta en el cálculo relacional se transforma a una en el álgebra relacional. Se sabe que puede existir más de una transformación. Por tanto, el enfoque seguido usualmente es empezar con una consulta algebraica y aplicar transformaciones para mejorarla.

**Localización de Datos**

La entrada a esta capa es una consulta algebraica definida sobre relaciones distribuidas. El objetivo de esta capa es localizar los datos de la consulta usando la información sobre la distribución de datos. Esta capa determina cuales fragmentos están involucrados en la consulta y transforma la consulta distribuida en una consulta sobre fragmentos.

**Optimización Global de Consultas**

Dada una consulta algebraica sobre fragmentos, el objetivo de esta capa es hallar una estrategia de ejecución para la consulta cercana a la óptima. La estrategia de ejecución para una consulta distribuida puede ser descrita con los operadores del álgebra relacional y con primitivas de comunicación para transferir datos entre nodos. Para encontrar una buena transformación se consideran las características de los fragmentos, tales como, sus cardinalidades. Un aspecto importante de la optimización de consultas es el ordenamiento de juntas, dado que algunas permutaciones de juntas dentro de la consulta pueden conducir a un mejoramiento de varios órdenes de magnitud. La salida de la capa de optimización global es una consulta algebraica optimizada con operación de comunicación incluidas sobre los fragmentos.

**Optimización Local de Consultas**

El trabajo de la última capa se efectúa en todos los nodos con fragmentos involucrados en la consulta. Cada subconsulta que se ejecuta en un nodo, llamada *consulta local*, es optimizada usando el esquema local del nodo. Hasta este momento, se pueden eligen los algoritmos para realizar las operaciones relacionales. La optimización local utiliza los algoritmos de sistemas centralizados.

4.6 Descomposición de consultas

Como se dijo en la sección anterior la descomposición de consultas consiste de cuatro pasos: normalización, análisis, simplificación y reestructuración. En esta sección se abundará más sobre cada uno de los pasos.

4.6.1 Normalización

La consulta de entrada puede ser arbitrariamente compleja dependiendo de las facilidades provistas por el lenguaje. El objetivo de la normalización es transformar una consulta a una forma normalizada para facilitar su procesamiento posterior. La normalización consiste de dos partes:

El análisis léxico y sintáctico. En esta parte se verifica la validez de la expresión que da origen a la consulta. Se verifica que las relaciones y atributos invocados en la consulta estén acordes con la definición en la base de datos. Por ejemplo, se verifica el tipo de los operandos cuando se hace la calificación.

Construcción de la forma normal. Existen dos tipos de formas normales. La forma normal conjuntiva es una conjunción de disyunciones como sigue:

$$(p_{11} \text{ Ú } p_{12} \text{ Ú } . . . \text{ Ú } p_{1n}) \text{ Ù } (p_{21} \text{ Ú } p_{22} \text{ Ú } . . . \text{ Ú } p_{2n}) \text{ Ù } . . . \text{ Ù } (p_{m1} \text{ Ú } p_{m2} \text{ Ú } . . . \text{ Ú } p_{mn})$$

donde *pij* es un predicado simple. Una consulta está en forma normal disyuntiva cuando se tiene una disyunción de conjunciones:

$$(p_{11} \text{ Ù } p_{12} \text{ Ù } . . . \text{ Ù } p_{1n}) \text{ Ú } (p_{21} \text{ Ù } p_{22} \text{ Ù } . . . \text{ Ù } p_{2n}) \text{ Ú } . . . \text{ Ú } (p_{m1} \text{ Ù } p_{m2} \text{ Ù } . . . \text{ Ù } p_{mn})$$

En cualquier forma normal, la expresión está libre de cuantificadores, existencial o universal, por lo que solo se consideran predicados simples. Existe un procedimiento para obtener la forma normal, conjuntiva o disyuntiva, de cualquier expresión en el cálculo de proposiciones. Para la aplicación que estamos considerando, la forma normal conjuntiva es más práctica puesto que incluye más operadores AND (Ù ) que operadores OR (Ú ). Típicamente, los operadores OR se transforman en uniones de conjuntos y los operadores AND se transforman en operadores de junta o selección.

**Ejemplo 4.3**. Considere la siguiente consulta en la base de datos de ingeniería que hemos utilizado a lo largo de estas notas.

"Encuentre los nombres de los empleados que han trabajado en el proyecto J1 12 o 24 meses"

La consulta expresada en SQL es:

**SELECT** ENAME

**FROM** E, G

**WHERE** E.ENO = G.ENO **AND** G.JNO = "J1" **AND** DUR = 12 OR DUR = 24

La consulta en forma normal conjuntiva es:

E.ENO = G.ENO Ù G.JNO = "J1" Ù (DUR = 12 Ú DUR = 24)

La consulta en forma normal disyuntiva es:

(E.ENO = G.ENO Ù G.JNO = "J1" Ù DUR = 12) Ú

(E.ENO = G.ENO Ù G.JNO = "J1" Ù DUR = 24)

En esta última forma, si se tratan las conjunciones en forma independiente se puede incurrir en trabajo redundante si no se eliminan las expresiones comunes.

..

### 4.6.2 Análisis

El análisis de consultas permite rechazar consultas normalizadas para los cuales no se requiere mayor procesamiento. Una consulta se puede rechazar si alguno de sus atributos o nombres de relación no están definidas en el esquema global. También se puede rechazar si las operaciones que se aplican a los atributos no son del tipo adecuado.

Se puede hacer también un análisis semántico. La consulta se puede rechazar si las componentes no contribuyen de ninguna forma a la generación del resultado. Dentro del cálculo relacional no es posible determinar la correctitud semántica de una consulta general. Sin embargo, es posible hacerlo para una clase importante de consultas relacionales, aquellas que no contienen disyunciones y negaciones. El análisis anterior se basa en la representación de la consulta como una gráfica, llamada la *gráfica de la consulta* o la *gráfica de conectividad*. En una gráfica de consulta, un nodo indica la relación resultante, y cualquier otro nodo representa la relación operante. Un arco entre dos nodos que no son resultados representa una junta, mientras que un arco cuyo nodo destino es una relación resultante representa una proyección. Más aún, un nodo no resultado puede ser etiquetado por un predicado de selección o auto-junta. Una subgráfica importante de la gráfica de conectividad es la gráfica de juntas, en la cual únicamente se consideran las juntas. La gráfica de juntas es particularmente importante durante la fase de optimización.

**Ejemplo 4.4**. Considere la siguiente consulta:

"Encuentre los nombres y responsabilidades de los programadores que han estado trabajando en el proyecto de CAD/CAM por más de tres años y el nombre de su administrador"

La consulta expresada en SQL es:

**SELECT** ENAME, RESP

**FROM** E, G, J

**WHERE** E.ENO = G.ENO **AND** G.JNO = J.JNO **AND** JNAME = "CAD/CAM"

**AND** DUR ³ 36 **AND** TITLE = "Programador"

La gráfica de la consulta anterior se presenta en la Figura 4.5a. En la Figura 4.5b se presenta la gráfica de juntas para la gráfica de la Figura 4.5a.

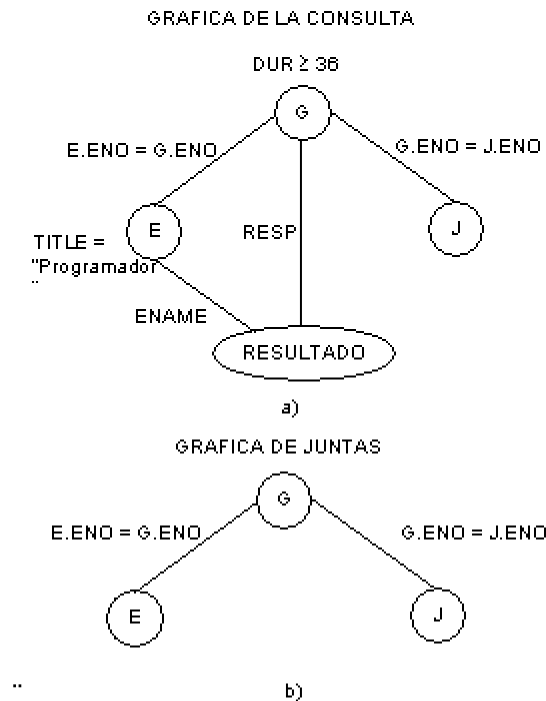



Figura 4.5. a) Gráfica de una consulta. b) Gráfica de juntas.

La gráfica de la consulta es útil para determinar la correctitud semántica de una consulta conjuntiva con múltiples variables sin negaciones. Tal consulta es semánticamente incorrecta si su gráfica no es conectada. En este caso, una o más subgráficas están desconectadas de la gráfica que contiene la relación RESULTADO.

**Ejemplo 4.5**. Considere la siguiente consulta:

**SELECT** ENAME, RESP

**FROM** E, G, J

**WHERE** E.ENO = G.ENO **AND** JNAME = "CAD/CAM"

**AND** DUR ³ 36 **AND** TITLE = "Programador"

La gráfica de la consulta anterior se presenta en la Figura 4.6, la cual se ve claramente que es no conectada.

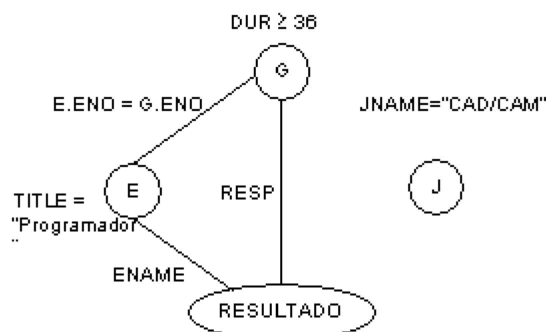



Figura 4.6. Gráfica de una consulta semánticamente incorrecta.

### 4.6.2 SIMPLIFICACION

La consulta en forma normal conjuntiva puede contener predicados redundantes. Una evaluación directa de la consulta con redundancia puede llevarnos a realizar trabajo duplicado. La redundancia puede ser eliminada aplicando sucesivamente las siguientes reglas de idempotencia:

1. *p* Ù *p* Û *p*
2. *p* *Ú* *p* Û *p*
3. *p* Ù *true* Û *p*
4. *p* Ú *false* Û *p*
5. *p* Ù *false* Û *false*
6. *p* Ú *true* Û *true*
7. *p* Ù Ø *p* Û *false*
8. *p* Ú Ø *p* Û *true*
9. $p_1$ Ù ($p_1$ Ú $p_2$) Û $p_1$
10. $p_1$ Ú ($p_1$ Ù $p_2$) Û $p_1$

**Ejemplo 4.6**. La siguiente consulta en SQL

```
SELECT TITULO

FROM E

WHERE (NOT (TITULO = "Programador"))

AND (TITULO = "Programador"

OR TITULO = "Ingeniero Eléctrico")

AND NOT (TITULO = "Ingeniero Eléctrico")

OR ENOMBRE = "J. Doe"
```

puede ser simplificada usando las reglas anteriores a

```
SELECT TITULO

FROM E

WHERE ENOMBRE = "J. Doe"
```

¨

### 4.6.2 REESTRUCTURACION

El último paso en la descomposición de consultas reescribe la consulta en el álgebra relacional. Esto se hace típicamente en los siguientes paso:

1. Una transformación directa del cálculo relacional en el álgebra relacional
2. Una reestructuración de la consulta en el álgebra relacional para mejorar la eficiencia

Por claridad es costumbre representar la consulta en el álgebra relacional por un árbol del álgebra relacional, el cual es un árbol en donde una hoja representa a una relación almacenada en la base de datos y un nodo no hoja es una relación intermedia producida por una operación del álgebra relacional.

La transformación de una consulta en el cálculo relacional en un árbol del álgebra relacional se puede hacer como sigue. Primero, se crea una hoja diferente para cada variable de tuplo diferente. En SQL, las hojas están disponibles de forma inmediata en la cláusula FROM. Segundo, el nodo raíz se crea como una operación de proyección involucrando a los atributos resultantes. Estos se encuentran en la cláusula SELECT de una consulta en SQL. Tercero, la calificación (WHERE) de una consulta se traduce a una secuencia apropiada de operaciones relacionales (select, join, union, etc.) yendo de las hojas a la raíz. La secuencia se puede dar directamente por el orden de aparición de los predicados y operadores.

**Ejemplo 4.7**. La consulta

"Encuentre los nombres de empleados diferentes de "J. Doe" que trabajaron en el proyecto de CAD/CAM por uno o dos años"

se puede expresar en SQL como sigue:

**SELECT** ENAME

**FROM** J, E, G

**WHERE** E.ENO = G.ENO

**AND** G.JNO = J.JNO

**AND** ENAME ¹ "J. Doe"

**AND** JNAME = "CAD/CAM"

**AND** (DUR = 12 OR DUR = 24)

Se puede mapear de manera directa al árbol del álgebra relacional de la Figura 4.7.

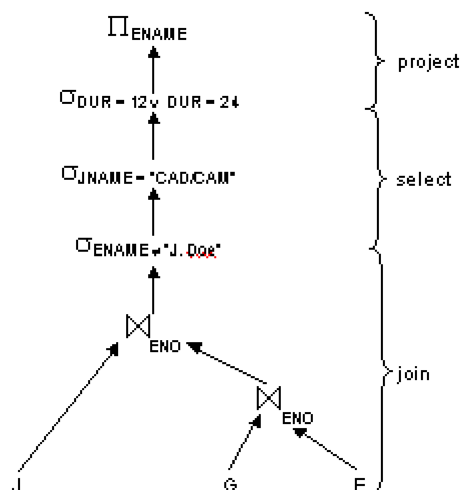



Figura 4.7. Ejemplo de un árbol del álgebra relacional.

Aplicando las reglas de transformación que se verán a continuación, muchos árboles diferentes se pueden encontrar por el procedimiento descrito antes. Las seis reglas de equivalencia más útiles, las cuales están relacionadas a las operaciones del álgebra relacional se presentan a continuación. En lo que sigue, $R$, $S$ y $T$ son relaciones donde $R$ se define sobre los atributos $A$ = { $A_1$, $A_2$, ..., $A_n$ } y $S$ se define sobre los atributos $B$ = { $B_1$, $B_2$, ..., $B_n$ }.

1. Conmutatividad de operaciones binarias:

   *R* ´ *S* = *S* ´ *R*

   *R* > < *S* = *S* > < *R*

   *R* È *S* = *S* È *R*

2. Asociatividad de operaciones binarias:

   *R* ´ (*S* ´ *T)* Û (*R* ´ *S*) ´ *T*

   *R* > < (*S* > < *T)* Û (*R* > < *S*) > < *T*

3. Idempotencia de operaciones unarias:

   P $_{A'}$ (P $_{A''}$ ($R$)) Û P $_{A'}$ ($R$)

   s $_{p1(A1)}$ (s $_{p2(A2)}$ ($R$)) Û s $_{p1(A1)\ Ù\ p2(A2)}$ ($R$)

   donde $R$[$A$] y $A'$ Í $A$, $A''$ Í $A$ y $A'$ Í $A$.

4. Conmutando selección con proyección

   P $_{A1,\ ...,\ An}$ (s $_{p(Ap)}$ ($R$)) Û P $_{A1,\ ...,\ An}$ (s $_{p(Ap)}$ (P $_{A1,\ ...,\ An,\ Ap}$ ($R$)))

5. Conmutando selección con operaciones binarias

   s $_{p(A)}$ ($R$ ´ $S$) Û (s $_{p(A)}$ ($R$)) ´ $S$

   s $_{p(Ai)}$ ($R$ > < $_{(Aj,\ Bk)}$ $S$) Û (s $_{p(Ai)}$ $R$) > < $_{(Aj,\ Bk)}$ $S$

   s $_{p(Ai)}$ ($R$ È $T$) Û s $_{p(Ai)}$ ($R$) È s $_{p(Ai)}$ ($T$)

   donde $A_i$ pertenece a $R$ y a $T$.

6. Conmutando proyección con operaciones binarias

   P $_{C}$ ($R$ ´ $S$) Û P $_{A'}$ ($R$) ´ P $_{B'}$ ($S$)

   P $_{C}$ ($R$ > < $_{(Aj,\ Bk)}$ $S$) Û P $_{A'}$ ($R$) > < $_{(Aj,\ Bk)}$ P $_{B'}$ ($S$)

   P $_{C}$ ($R$ È $S$) Û P $_{C}$ ($R$) È P $_{C}$ ($T$)

donde *R*[*A*] y *S*[*B*]; *C* = *A*' È *B*' donde *A*' Í *A* y *B*' Í *B*.

**Ejemplo 4.8**. En la Figura 4.8 se presenta un árbol equivalente al mostrado en la Figura 4.7. La reestructuración del árbol de la Figura 4.7 se presenta en la Figura 4.9. El resultado es bueno en el sentido de que se evita el acceso repetido a la misma relación y las operaciones más selectivas se hacen primero. Sin embargo, este árbol aún no es óptimo. Por ejemplo, la operación de selección E no es muy útil antes de la junta dado que no reduce grandemente el tamaño de la relación intermedia.

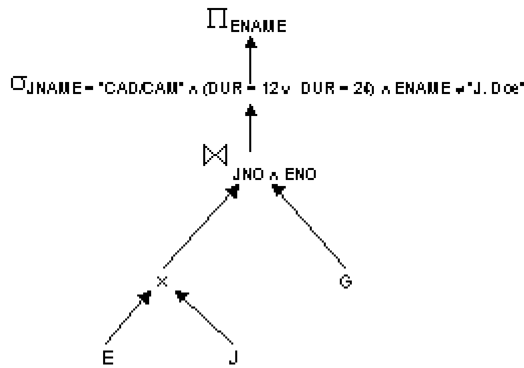



Figura 4.8. Arbol equivalente al de la Figura 4.7.

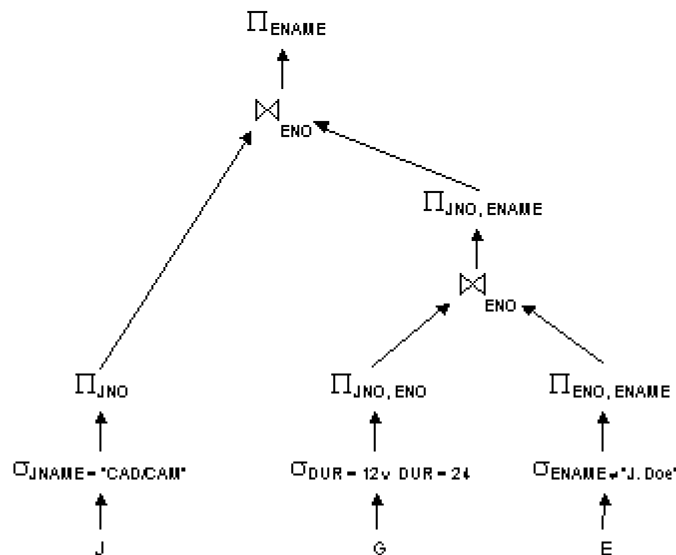



Figura 4.9. Arbol reestructura a partir del de la Figura 4.7.

### 4.7 Localización de datos distribuidos

En la sección anterior se presentaron técnicas generales para la descomposición y reestructuración de consultas expresadas en el cálculo relacional. Esas técnicas globales se aplican tanto a bases de datos centralizadas como a distribuidas; no toman en cuenta la distribución de datos. Este es el papel de la capa de localización, la cual traduce una consulta hecha sobre relaciones globales a una consulta algebraica expresada en fragmentos físicos. La localización utiliza información almacenada en el esquema de fragmentación. Por simplicidad en esta sección no se considera el caso de fragmentos replicados.

La fragmentación de una relación se define a través de las reglas de fragmentación, las cuales pueden ser expresadas como consultas relacionales. Una relación global puede ser reconstruida aplicando las reglas de reconstrucción y derivando un programa en el álgebra relacional cuyos operandos son los fragmentos. A este programa se le conoce como *programa de localización*. Una forma simple de localizar una consulta distribuida es generar una consulta donde cada relación global es sustituida por su programa de localización. Esto puede ser visto como el reemplazo de las hojas del árbol del álgebra relacional de la consulta distribuida con subárboles que corresponden a los programas de localización. A la consulta obtenida por esta forma se le conoce como una *consulta genérica*.

En general, el enfoque anterior puede ser ineficiente dado que varias simplificaciones y reestructuraciones de la consulta genérica aún pueden ser realizadas. En lo que sigue de esta sección, por cada tipo de fragmentación se presentan técnicas de reducción que general consultas simples y optimizadas.

### 4.7.1 Reducción para fragmentación horizontal primaria

La fragmentación horizontal distribuye una relación basada en predicados de selección. El ejemplo siguiente será usado a lo largo de esta sección.

**Ejemplo 4.9**. La relación E(ENO, ENOMBRE, TITULO) puede ser dividida en tres fragmentos horizontales $E_1$, $E_2$ y $E_3$, definidos como sigue:

$$E_1 = \sigma_{ENO \leq "E3"} (E)$$

$$E_2 = \sigma_{"E3" < ENO \leq "E6"} (E)$$

$$E_3 = \sigma_{ENO > "E6"} (E)$$

El programa de localización para fragmentación horizontal es la unión de los fragmentos. Aquí se tiene que:

$$E = E_1 \cup E_2 \cup E_{13}$$

La relación G puede ser dividida en dos fragmentos horizontales $G_1$ y $G_2$ definidos como sigue:

$$G_1 = \sigma_{ENO \leq "E3"} (G)$$

$$G_2 = \sigma_{ENO > "E3"} (G)$$

El programa de localización para G es la unión de los fragmentos. Aquí se tiene que:

$$G = G_1 \cup G_2$$

El árbol genérico se presenta en la Figura 4.10.

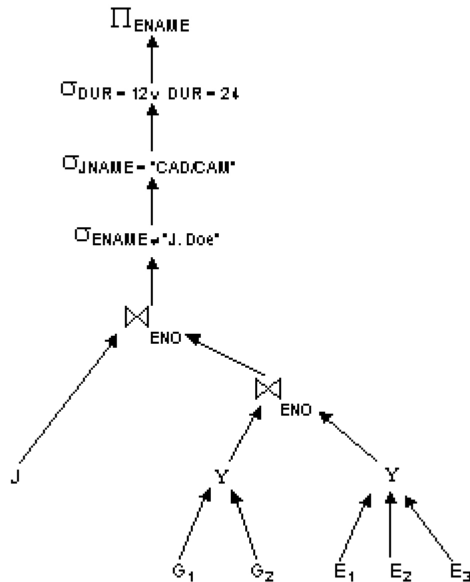



Figura 4.10. Arbol genérico para el ejemplo 4.9.

**Reducción con selección**

Dada una relación *R* que ha sido fragmentada horizontalmente como $R_1$, $R_2$, ..., $R_w$, donde $R_j$ = s $_{pj}$( *R* ), la regla puede ser formulada como sigue

**Regla 1**: s $_{pj}$( $R_j$ ) = f si " *x* en *R*: Ø ($p_i(x)$ Ù $p_j(x)$)

donde $p_i(x)$ y $p_j(x)$ son predicados de selección, *x* denota a un tuplo, y *p*( *x* ) denota que el predicado *p* se satisface para *x*.

**Ejemplo 4.10**. Considere la siguiente consulta

```
SELECT *

FROM E

WHERE ENO = "E5"
```

Aplicando el enfoque directo para localizar E a partir de $E_1$, $E_2$ y $E_3$, se obtiene la consulta genérica de la Figura 4.11a. Conmutando la selección con la operación de unión, es fácil detectar que el predicado de selección contradice los predicados de $E_1$ y $E_3$, produciendo relaciones vacías. La consulta reducida es simplemente aplicada a $E_2$ como se muestra en la Figura 4.11b.

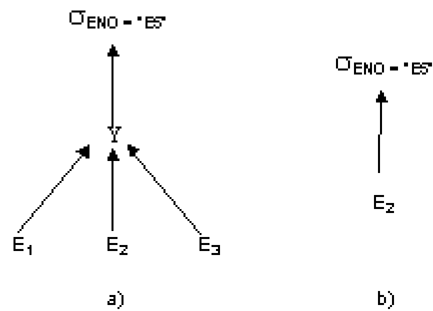



Figura 4.11. Reducción para fragmentación horizontal con selección.

**Reducción con junta**

Juntas en relaciones fragmentadas horizontalmente pueden ser simplificadas cuando las relaciones juntadas están fragmentadas de acuerdo al atributo de la junta. La simplificación consiste en distribuir las juntas sobre las uniones y eliminar juntas inútiles. La distribución de una junta sobre una unión puede ser establecida como

$(R_1$ È $R_2) > < R_3 = (R_{1> <} R_3)$ È $(R_2 > < R_3)$

donde $R_i$ son fragmentos. Con esta transformación las uniones pueden ser movidas hacia arriba en el árbol de consulta de manera que todas las posibles juntas de fragmentos son exhibidas. Juntas inútiles de fragmentos pueden ser determinadas cuando los predicados de fragmentos juntados son contradictorios. Suponga que los fragmentos $R_i$ y $R_j$ están definidos de acuerdo a los predicados $p_i$ y $p_j$, respectivamente, sobre el mismo atributo, la regla de simplificación puede formularse como sigue:

**Regla 2**: $R_{1> <} R_3$ = f si " $x$ en $R_i$, " $y$ en $R_j$: Ø ($p_i(x)$ Ù $p_j(y)$)

**Ejemplo 4.11**. Considere la fragmentación de la relación G del Ejemplo 4.11 junto con la fragmentación de E. $E_1$ y $G_1$ están definidos de acuerdo al mismo predicado. Más aún, el predicado que define a $G_2$ es la unión de los predicados que definen $E_2$ y $E_3$. Ahora considere la consulta con junta

```
SELECT *

FROM E, G

WHERE ENO = G.ENO
```

La consulta genérica equivalente se presenta en la Figura 4.12a. La consulta reducida se puede observar en la Figura 4.12b.

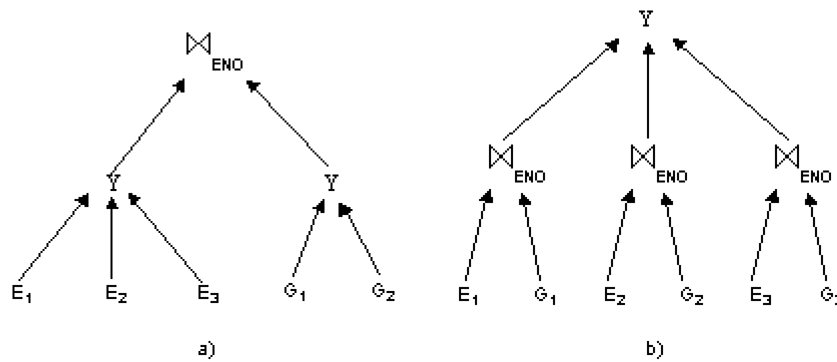



Figura 4.12. Reducción para fragmentación horizontal con junta.

### 4.7.2 Reducción para fragmentación vertical

La fragmentación vertical distribuye una relación de acuerdo a los atributos de proyección. Dado que el operador de reconstrucción para la fragmentación vertical es la junta, el programa de localización para una relación fragmentada verticalmente consiste de la junta de los fragmentos sobre el atributo común.

**Ejemplo 4.12**. Considere la relación E dividida en dos fragmentos verticales donde el atributo ENO sirve como atributo común.

$E_1 = P_{ENO,ENOMBRE} (E)$

$E_2 = P_{ENO,TITULO} (E)$

El programa de localización es

$E = E_1 > < E_2$

..

La reducción de consultas sobre fragmentos verticales se hace determinando relaciones intermedias inútiles y eliminando los subárboles que las producen. Las proyecciones sobre fragmentos verticales que no tienen atributos en común con los atributos de proyección (excepto la llave de la relación) producen relaciones inútiles aunque probablemente no vacías.

Dada una relación *R* definida sobre los atributos $A = \{ A_1, A_2, ..., A_n \}$, la cual se fragmenta verticalmente como $R_i = P_{A'} (R)$, donde *A*' Í *A*, la regla para determinar relaciones intermedias inútiles se puede formular como sigue:

**Regla 3**: $P_{D,K} (R)$ es inútil si el conjunto de atributos de proyección *D* no está en *A*'.

**Ejemplo 4.13**. Considere de nuevo la relación E dividida en fragmentos verticales como en el Ejemplo 4.12. Considere también la siguiente consulta en SQL:

```
SELECT ENAME

FROM E
```

$E_1 = P_{ENO,ENOMBRE}\ (E)$

$E_2 = P_{ENO,TITULO}\ (E)$

La consulta genérica equivalente se presenta en la Figura 4.13a. Conmutando la proyección con la junta, se puede ver que la proyección sobre $E_2$ es inútil dado que ENOMBRE no está en $E_2$. Por lo tanto, la proyección necesita aplicarse únicamente a $E_1$, como se presenta en la Figura 4.13b.

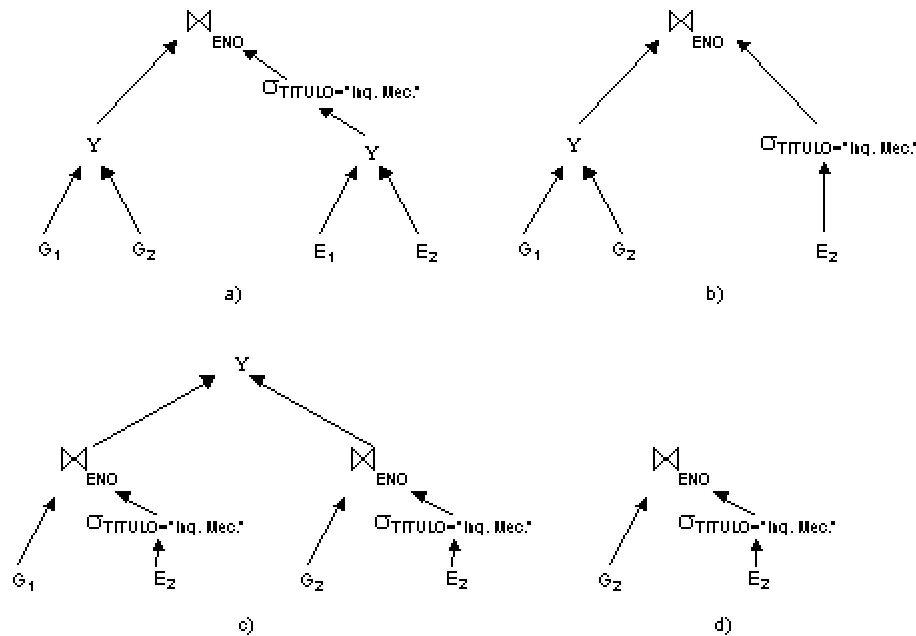



Figura 4.13. Reducción para fragmentación vertical.

### 4.7.3 Reducción para fragmentación horizontal derivada

Si una relación *R* es sometida a una fragmentación horizontal derivada con respecto a *S*, los fragmentos de *R* y *S* que tienen el mismo valor del atributo para la junta se localizan en el mismo nodo. Así, *S* puede ser fragmentado de acuerdo al predicado de selección. Dado que los tuplos de *R* se colocan de acuerdo a los tuplos de *S*, la fragmentación horizontal derivada debe ser usada solo para relaciones uno-a-muchos de la forma *S* *R*, donde un tuplo de *S* se asocia con *n* tuplos de *R*, pero un tuplo de *R* se asocia exactamente con uno de *S*.

**Ejemplo 4.14**. Dado una relación uno-a-muchos de E a G, la relación G[ENO, JNO, RESPONSABLE, DUR] se puede fragmentar indirectamente de acuerdo a las siguientes reglas:

$G_1 = G > <_{ENO} E_1$

$G_2 = G > <_{ENO} E_2$

La relación E es fragmentada horizontalmente de la siguiente manera:

$E_1 = s_{TITULO=\text{"Programador"}}\ E$

$E_2 = s_{TITULO^1\ \text{"Programador"}}\ E$

El programa de localización para G es

$G = G_1$ È $G_2$

Las consultas en fragmentos derivados también se pueden reducir. Dado que este tipo de fragmentación es útil para optimizar consultas con juntas, una transformación útil es distribuir las juntas sobre las uniones (usadas en los programas de localización) y aplicar la regla 2 presentada antes. Ya que las reglas de fragmentación indican como se asocian los tuplos, ciertas juntas producirán relaciones vacías si los predicados de la fragmentación son inconsistentes.

**Ejemplo 4.15**. Considere la siguiente consulta en SQL sobre la fragmentación definida en el Ejemplo 4.14:

**SELECT** *

**FROM** E, G

**WHERE** G.ENO = E.ENO AND TITLE = "Ingeniero Mecánico"

La consulta genérica de se presenta en la Figura 4.14a. Aplicando primero la selección sobre los fragmentos $E_1$ y $E_2$, el predicado de la selección es inconsistente con el de $E_1$, por lo tanto, E1 puede ser eliminado y la consulta se reduce a la mostrada en la Figura 4.14b. Ahora se distribuyen las juntas sobre las uniones produciendo el árbol de la Figura 4.14c. El subárbol de la izquierda junta los fragmentos $G_1$ y $E_2$, sin embargo, sus predicados son inconsistentes (TITULO = "Programador" de $G_1$ es inconsistente con TITULO ¹ "PROGRAMADOR" en $E_2$). Así, el subárbol de la izquierda produce una relación vacía por lo que puede ser eliminado obteniendo la consulta reducida de la Figura 4.14d.

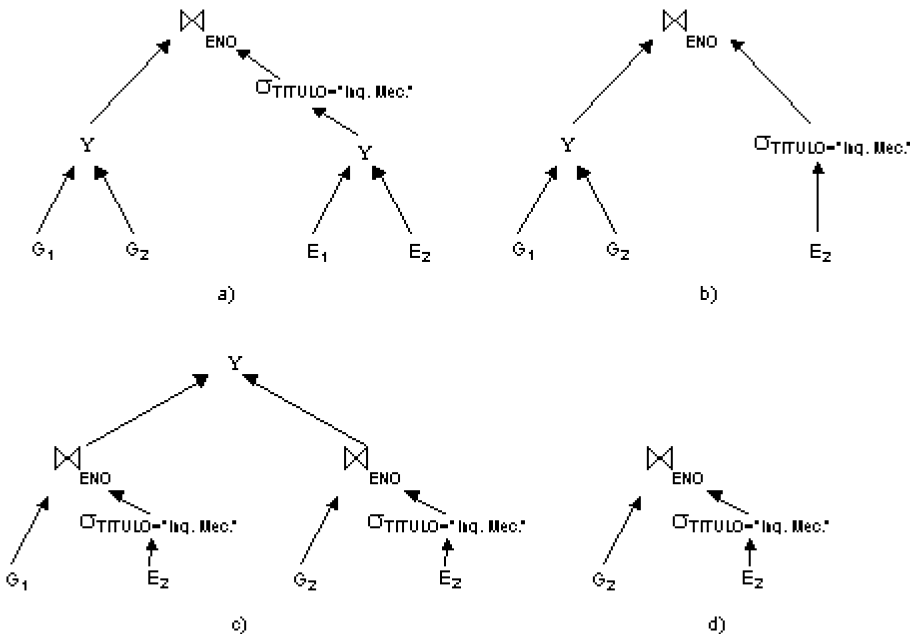



Figura 4.14. Reducción para fragmentación horizontal derivada.

### 4.7.3 Reducción para fragmentación híbrida

La fragmentación híbrida se obtiene combinando fragmentación horizontal y vertical. El objetivo de la fragmentación híbrida es ejecutar de manera eficiente consultas que involucran selección, proyección y junta. El programa para fragmentación híbrida utiliza uniones y juntas de fragmentos. Las consultas en fragmentos híbridos se pueden reducir combinando las reglas usadas para fragmentación horizontal primaria, fragmentación vertical y fragmentación horizontal derivada. Estas se pueden resumir de la manera siguiente:

1. Remover las relaciones vacías generadas para por selecciones contradictorias en fragmentos horizontales.
2. Remover las relaciones intermedias inútiles generadas por proyecciones en fragmentos verticales.
3. Distribuir juntas sobre uniones a fin de aislar y remover juntas inútiles.

**Ejemplo 4.16**. Considere la siguiente fragmentación híbrida de la relación E:

$E_1 = s_{\text{ENO £ "E4"}} (P_{\text{ENO,ENOMBRE}} (E))$

$E_1 = s_{\text{ENO > "E4"}} (P_{\text{ENO,ENOMBRE}} (E))$

$E_3 = P_{\text{ENO,TITULO}} (E)$

El programa de localización de para la relación E es

$E = (E_1 \text{ È } E_2)_{> < \text{ENO}} E_3$

Considere ahora la siguiente consulta en SQL

```
SELECT ENAME

FROM E

WHERE E.ENO = "E5"
```

La consulta genérica se presenta en la Figura 4.15a. Si se mueve hacia abajo la operación de selección se puede eliminar $E_1$. Más aún, si, una vez eliminando $E_1$, se mueve hacia abajo la operación de proyección, entonces, se puede eliminar $E_3$. Así, la junta ya no es necesaria y se obtiene la consulta reducida de la Figura 4.15b.

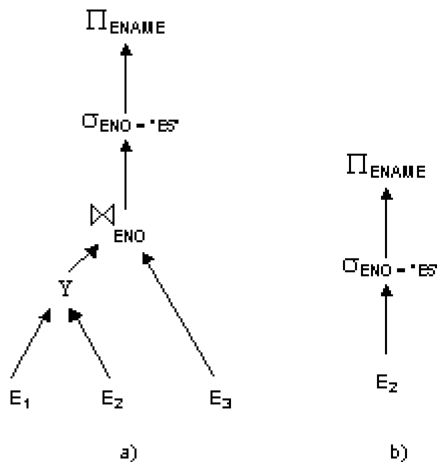



Figura 4.15. Reducción para fragmentación horizontal derivada.

### 4.8 Optimización global de consultas

La optimización global de consultas es la tercera etapa del procesamiento de consultas distribuidas de acuerdo a la Figura 4.4. Dada una consulta algebraica sobre fragmentos, el objetivo de esta capa es hallar una estrategia de ejecución para la consulta cercana a la óptima. La salida de la capa de optimización global es una calendarización de una consulta optimizada en el álgebra relacional la cual indica el orden en que se ejecutan los operadores e incluye operaciones de comunicación sobre los fragmentos.

Ya que la selección del ordenamiento óptimo de operadores para una consulta es computacionalmente intratable, el objetivo del optimizador es encontrar la mejor estrategia, no necesariamente óptima, para ejecutar la consulta. La selección de la "mejor" estrategia requiere, por lo general, predecir los costos de ejecución de diferentes alternativas para ejecutar la consulta. El costo de ejecución se expresa como la combinación pesada de los costos de entrada/salida, de CPU y de comunicación.

#### 4.8.1 Definiciones básicas

Antes de empezar los conceptos involucrados en la optimización de consultas, se revisarán brevemente algunos conceptos básicos en el proceso de optimización. En un proceso de optimización es necesario optimizar (minimizar o maximizar) una función objetivo, en nuestro caso llamada *función de costo*. El proceso de optimización selecciona el mejor miembro *x* de un conjunto que optimiza la función de costo. A este conjunto de posibles soluciones se le conoce como el conjunto solución. Dada la forma en que trabajan los algoritmos de optimización, al conjunto solución también se le conoce como el *espacio de búsqueda*. Los algoritmos, llamados de búsqueda, se mueven de elemento a elemento en este espacio hasta encontrar una buena solución. Existen diversas técnicas para el desarrollo de algoritmos de búsqueda: búsqueda ávida, programación dinámica, algoritmos heurísticos, mejoramiento iterativo, recocido simulado, algoritmos genéticos, búsqueda tabú, etc.

#### 4.8.2 Modelo de costo

El costo de una estrategia de ejecución distribuida se puede expresar con respecto al costo total (tiempo de ejecución) o al tiempo de respuesta. El costo total es la suma de todas sus componentes (I/O, CPU y comunicación). El tiempo de respuesta es el tiempo transcurrido desde el inicio de la consulta hasta su terminación. Ambas estrategias de optimización son diferentes. Mientras que en la función de costo total se trata de reducir el costo de cada componente haciendo que éstas sean tan rápidas como sea posible,

en la función del tiempo de respuesta se trata de hacer tantas cosas en forma simultánea (paralela) como sea posible lo que puede conducir a un incremento en el tiempo total de ejecución de la consulta.

**Costo Total**

El costo total es la suma de todos los factores de costo y puede expresarse como

*Costo total* = *costo de* I/O + *costo de* CPU + *costo de comunicación*

en donde,

*costo de* CPU = *costo de una instrucción* * *no. de instrucciones*

*costo de* I/O = *costo unitario de una operación de* I/O *a disco* * *no. de accesos*

*costo de comunicación* = (*tiempo de iniciación* + *tiempo de transmisión*) *

*no. de mensajes*

Los diferentes factores pueden tener pesos diferentes dependiendo del ambiente distribuido en el que se trabaje. Por ejemplo, en las redes de área amplia (WAN), normalmente el costo de comunicación domina dado que hay una velocidad de comunicación relativamente baja, los canales están saturados y el trabajo adicional requerido por los protocolos de comunicación es considerable. Los costos de iniciación y transmisión de mensajes en una WAN son relativamente altos relativos a los tiempos de procesamiento local. Por otra parte, un rango común entre el costo de comunicación y el costo de I/O es 20:1. Así, los algoritmos diseñados para trabajar en una WAN, por lo general, ignoran los costos de CPU y de I/O. En redes de área local (LAN), por otro lado, el costo de comunicación no es tan dominante, así que se consideran los tres factores con pesos variables.

**Tiempo de Respuesta**

El tiempo de respuesta es el tiempo transcurrido desde el inicio de la consulta hasta su terminación y se puede expresar como

*Costo total* = *costo de* I/O + *costo de* CPU + *costo de comunicación*

en donde,

*costo de* CPU = *costo de una instrucción* * *no. de instrucciones secuenciales*

*costo de* I/O = *costo unitario de una operación de* I/O *a disco* *

*no. de accesos secuenciales*

*costo de comunicación* = (*tiempo de iniciación* + *tiempo de transmisión*) *

*no. de mensajes secuenciales*

**Ejemplo 4.17**. Considere el sistema distribuido mostrado en la Figura 4.16, en la cual se procesa la respuesta a la consulta en el nodo 3 con datos que provienen de los nodos 1 y 2. Por simplicidad se considera únicamente el costo de comunicación. Para este caso,

costo total = 2 * tiempo de iniciación + tiempo unitario de transmisión * ($x$ + $y$)

Por otro lado, el tiempo de respuesta es

tiempo de respuesta = max{ tiempo para enviar de 1 a 3,

tiempo para enviar de 2 a 3 }

donde

tiempo para enviar de 1 a 3 = tiempo de iniciación +

tiempo unitario de transmisión * $x$

tiempo para enviar de 2 a 3 = tiempo de iniciación +

tiempo unitario de transmisión * $y$

Figura 4.16. Ejemplo de las transferencias de datos para una consulta.

### 4.8.2 Estadísticas de la base de datos

El factor principal que afecta la eficiencia de la ejecución de una estrategia es el tamaño de las relaciones intermedias que son producidas durante la ejecución. Cuando una operación subsecuente se localiza en un nodo diferente, la relación intermedia debe ser transmitida sobre la red. Por lo tanto, es de un interés especial el poder estimar el tamaño de las relaciones intermedias para poder minimizar el tamaño de las transferencias de datos. Esta estimación se basa en información estadística acerca de las relaciones básicas para predecir las cardinalidades de los resultados de las operaciones relacionales. Para un relación $R$ definida sobre los atributos $A = \{ A_1, A_2, ..., A_n \}$ y fragmentada como $R_1, R_2, ..., R_r$, los datos estadísticos son típicamente:

1. La longitud de cada atributo: *length*( $A_i$ )
2. El número de valores distintos para cada atributo en cada fragmento: card( P $_{Ai}$, $R_j$ )
3. Los valores máximo y mínimo en el dominio de cada atributo: *min*( $A_i$ ), max( $Ai$ )
4. Las cardinalidades de cada dominio: *card*( *dom*[$A_i$] )
5. Las cardinalidades de cada fragmento: *card*( *dom*[$R_j$] )

En algunas ocasiones es útil aplicar el *factor de selectividad* al realizar una junta. Este se define como sigue:

$$SF_{\bowtie}(R,S) = \frac{card(R \bowtie S)}{card(R) * card(S)}$$

El tamaño de una relación intermedia $R$ es

*size*( *R* ) = *card*( *R* ) * *length*( *R* )

donde *length*(*R*) es la longitud en bytes de un tuplo de *R* calculada a partir de sus atributos. La estimación de *card*( *R* ), el número de tuplos en *R*, requiere de las fórmulas de la siguiente sección.

**Cardinalidades de las Relaciones Intermedias**

Las cardinalidad de las operaciones son las siguientes:

**Selección**: *card*( s $_F$(*R*)) = $SF_S(F)$ * *card*(*R*)

donde $SF_S(F)$ es el factor de selección de la fórmula *F* calculada como sigue, donde $p(A_i)$ y $p(A_j)$ indican predicados sobre los atributos $A_i$ y $A_j$, respectivamente.

$$SF_S(A = valor) = \frac{1}{card(\Pi_A(R))}$$

$$SF_S(A > valor) = \frac{max(A) - valor}{max(A) - min(A)}$$

$$SF_S(A < valor) = \frac{valor - min(A)}{max(A) - min(A)}$$

$$SF_S(p(A_i) \wedge p(A_j)) = SF_S(p(A_i)) * SF_S(p(A_j)$$

$$SF_S(p(A_i) \vee p(A_j)) = SF_S(p(A_i)) + SF_S(p(A_j) - (SF_S(p(A_i)) * SF_S(p(A_j))$$

$$SF_S(A \in \{valores\}) = SF_S(A = valor) * card(\{valores\})$$

**Proyección**: *card*( P $_A$(*R*)) = *card*(*R*)

**Producto Cartesiano**: *card*( *R* ´ *S*) = *card*(*R*) * *card*(*S*)

**Unión**:

cota superior: *card*( *R* È *S*) = *card*(*R*) + *card*(*S*)

cota inferior: *card*( *R* È *S*) = max{*card*(*R*), *card*(*S*)}

**Diferencia de conjuntos**:

cota superior: *card*( *R* - *S*) = *card*(*R*)

cota inferior: *card*( *R* - *S*) = 0

**Junta**:

caso general: $card( R > < S) = SF> < * card(R) * card(S)$

caso especial: $A$ es una llave de $R$ y $B$ es una llave externa de $S$;

$A$ es una llave externa de $R$ y $B$ es una llave de $S$

$card( R > <_{=B} S) = card(R)$

**Semijunta**: $card( R > <_{A} S) = SF> < (S.A) * card(R)$

donde $SF> < ( R > <_{A} S) = SF> < (S.A) = \frac{card(\Pi_A(S))}{card(dom[A])}$

### 4.8.3 Optimización centralizada de consultas

En esta sección se revisa el proceso de optimización de consultas para ambientes centralizados. Esta presentación es un prerequisito para entender la optimización de consultas distribuidas por tres razones. Primero, una consulta distribuida se transforma a varias consultas locales cada una de las cuales se procesa en forma centralizadas. Segundo, las técnicas para optimización distribuida son, por lo general, extensiones de técnicas para optimización centralizada. Finalmente, el problema de optimización centralizada es mucho más simple que el problema distribuido.

Las técnicas de optimización más populares se deben a dos sistemas de bases de datos relacionales: INGRES y System R. Ambos sistemas tienen versiones para sistemas distribuidos pero sus técnicas difieren sustancialmente. Por un lado, INGRES utiliza una técnica dinámica la cual es ejecutada por un intérprete. Por otra parte, System R utiliza un algoritmo estático basado en búsqueda exhaustiva. Los sistemas más comerciales utilizan variantes de la búsqueda exhaustiva por su eficiencia y compatibilidad con la compilación de consultas.

Por brevedad, en esta sección se revisará brevemente el esquema de optimización de INGRES. Posteriormente, se revisará con mayor detenimiento el esquema de System R.

#### Algoritmo de INGRES

INGRES usa un algoritmo de optimización dinámico que recursivamente divide una consulta en el cálculo relacional en piezas más pequeñas. Combina dos fases de la descomposición cálculo-álgebra relacional y optimización. No requiere información estadística de la base de datos. Las dos fases generales del algoritmo son:

1. Descompone cada consulta con múltiples variables en una secuencia de consultas con una sola variable común.
2. Procesa cada consulta mono-variable mediante un procesador de consultas de una variable el cual elige, de acuerdo a algunas heurísticas, un plan de ejecución inicial. Posteriormente, ordena las operaciones de la consulta considerando los tamaños de las relaciones intermedias.

La primera parte se hace en tres pasos como se indica a continuación:

- Reemplaza una consulta $q$ con $n$ variables en una seria de consultas

$q_1$ ® $q_2$ ® ... ® $q_n$

donde $q_i$ usa el resultado de $q_{i-1}$.

- La consulta $q$ se descompone en $q'$ ® $q''$, donde $q'$ y $q''$ tienen una variable en común la cual es el resultado de $q'$.

- Se reemplaza el valor de cada tuplo con los valores actuales y se simplifica la consulta

$q( V_1, V_2, ..., V_n)$ ® $(q'( t_1, V_2, ..., V_n), t_1$ Î $R)$

**Ejemplo 4.17**. Para ilustrar el paso de desacoplamiento del algoritmo de INGRES considere la siguiente consulta:

"Encuentre los nombres de los empleados que trabajan en el proyecto CAD/CAM"

Esta consulta se puede expresar en SQL de la siguiente forma:

$q_1$: **SELECT** E.ENOMBRE

**FROM** E, G, J

**WHERE** E.ENO = G.ENO **AND** G.JNO = J.JNO **AND** JNAME = "CAD/CAM"

$q_1$ es reemplazada por $q_{11}$ seguida de $q'$ en donde JVAR es una relación intermedia.

$q_{11}$: **SELECT** J.JNO **INTO** JVAR

**FROM** J

**WHERE** JNAME = "CAD/CAM"

$q'$: **SELECT** E.ENOMBRE

**FROM** E, G, JVAR

**WHERE** E.ENO = G.ENO **AND** G.JNO = JVAR.JNO

La aplicación sucesiva de este paso a $q'$ puede generar

$q_{12}$: **SELECT** G.ENO **INTO** GVAR

**FROM** G, JVAR

**WHERE** G.JNO = JVAR.JNO

$q_{13}$: **SELECT** E.ENOMBRE

**FROM** G, GVAR

**WHERE** E.ENO = GVAR.ENO

Así, $q_1$ ha sido reducido a la secuencia de consultas $q_{11}$ ® $q_{12}$ ® $q_{13}$. La consulta $q_{11}$ es monovariable. Sin embargo, las consultas $q_{12}$ y $q_{13}$ no son monovariables. Veamos como transformar $q_{13}$. La relación definida por la variable GVAR es sobre un solo atributo (ENO). Supongamos que contiene únicamente dos tuplos: <E1> y <E2>. La sustitución de GVAR genera dos consultas de una sola variable:

$q_{131}$: **SELECT** E.ENOMBRE

**FROM** E

**WHERE** E.ENO = "E1"

$q_{132}$: **SELECT** E.ENOMBRE

**FROM** E

**WHERE** E.ENO = "E2"

## Algoritmo de System R

System R ejecuta una optimización de consultas estática basada en búsqueda exhaustiva del espacio solución. La entrada del optimizador es un árbol del álgebra relacional que resulta de una consulta en SQL. La salida es un plan de ejecución que implementa el árbol del álgebra relacional "óptimo".

El optimizador asigna un costo a cada árbol candidato y obtiene aquel con costo menor. Los árboles candidatos se obtienen permutando el orden de las juntas de las *n* relaciones de una consulta usando las reglas de conmutatividad y asociatividad. Para limitar el costo de optimización, el número de alternativas se reduce utilizando programación dinámica. El conjunto de estrategias alternativas se construye de forma dinámica, de manera que, cuando dos juntas son equivalentes por conmutatividad, se queda solo con la de costo más bajo. Más aún, cada vez que aparecen productos Cartesianos se trata de eliminar la estrategia correspondiente.

El costo de una estrategia candidato es una combinación pesada de costos de I/O y CPU. La estimación de tales costos (en tiempo de compilación) se basa en un modelo de costo que proporciona una fórmula de costo para cada operación de bajo nivel. Para la mayoría de las operaciones, esas fórmulas de costo se basan en las cardinalidades de los operandos. La información de las cardinalidades se obtiene del mismo System R. La cardinalidad de las relaciones intermedias se estima de acuerdo a los factores de selectividad de las operaciones.

El algoritmo de optimización consiste de dos grandes pasos:

1. Las consultas simples se ejecutan de acuerdo al mejor camino de acceso (basada en un predicado de selección).
2. Para cada relación *R*, se estima el mejor ordenamiento de juntas, en donde *R* se accesa primero usando el mejor método de acceso a una sola relación. Así, se determinan todos los posibles ordenamientos de juntas, se determina el costo de cada ordenamiento y se elige aquel con el costo mínimo.

Para la junta de dos relaciones, la relación cuyos tuplos se leen primero se llama *externa*, mientras que la otra, cuyos tuplos se encuentran de acuerdo a los valores obtenidos de la relación externa, se llama relación *interna*. Una decisión importante es determinar el camino de acceso menos costoso a la relación interna.

Al considerar las juntas, hay dos algoritmos disponibles. El primero, llamado de *ciclos anidados*, compone el producto de las dos relaciones y se puede observar mediante el siguiente pseudocódigo.

**Para cada** tuplo de la relación externa (con cardinalidad $n_1$)

**Para cada** tuplo de la relación interna (con cardinalidad $n_2$)

junta los dos tuplos si el predicado de la junta es verdadero

**end**

**end**

Claramente, este método tiene complejidad $n_1 * n_2$.

El segundo método, llamado *junta-mezcla*, consiste en mezclar dos relaciones ordenadas sobre el atributo de la junta. Los índices del atributo de la junta se pueden usar como caminos de acceso. Si el criterio de junta es igualdad, el costo de juntar dos relaciones de $n_1$ y $n_2$ páginas, respectivamente, es proporcional a $n_1 + n_2$.

**Ejemplo 4.18**. Considere la consulta $q_1$ del Ejemplo 4.17. La gráfica de juntas de se muestra en la Figura 4.17. Suponga que se tienen los siguientes índices:

E tiene un índice en ENO

G tiene un índice en GNO

J tiene un índice en JNO y un índice en JNOMBRE

Supongamos que se seleccionan los siguientes caminos de acceso mejores para cada relación:

E: recorrido secuencial (ya que no hay selección sobre E)

G: recorrido secuencial (ya que no hay selección sobre G)

J: índice sobre JNOMBRE (ya que hay una selección en J basada en JNOMBRE)

La construcción dinámica del árbol de estrategias alternativas se presenta en la Figura 4.18. Note que el número de ordenamientos posibles de juntas es 3! los cuales son:

E > < G > < J

G > < J > < E

J > < E > < G

J > < G > < E

G > < E > < J

E ´ J > < G

J ´ E > < G

La operación marcadas como "podadas" se eliminan en forma dinámica. El primer nivel del árbol indica los mejores caminos de acceso a una sola relación. El segundo nivel indica, el mejor método de junta con cualquier otra relación. Las estrategias que utilizan el pruducto Cartesiano son inmediatamente podadas. Suponga que (E > < G) y (G > < J) tienen un costo más alto que (G > < E) y (J > < G), respectivamente. Así ellas pueden ser podadas ya que existe un orden equivalente con costo menor. Las dos posibilidades restantes están dadas en el tercer nivel del árbol. El mejor orden de las juntas es el de costo menor entre ((G > < E) > < J) y ((J > < G) > < E). El último es el único que tienen un índice útil en el atributo de selección y acceso directo a los tuplos de junta entre G y E. Por lo tanto, éste es elegido con el siguiente método de acceso:

Selecciona J usando el índice sobre JNOMBRE

Junta con G usando el índice sobre JNO

Junta con E usando el índice sobre ENO

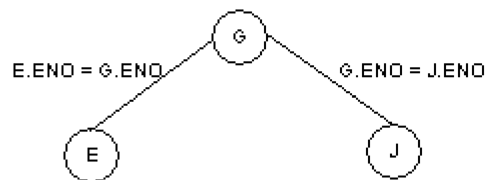



Figura 4.17. Gráfica de juntas.

### 4.8.4 Ordenamiento de juntas para consultas fragmentadas

Como se puede apreciar en la sección anterior, el ordenamiento de juntas es un aspecto importante para la optimización de consultas centralizadas. El ordenamiento de las juntas en ambientes distribuidos es aún más importante ya que las juntas entre fragmentos pueden incrementar los costos de comunicación.

Consideremos inicialmente el caso más simple de transferencia de operandos en una junta simple. La consulta es *R* > < *S* en donde *R* y *S* son relaciones almacenadas en nodos diferentes. La elección obvia es transferir la relación más pequeña al nodo con la relación más grande como se ilustra en la Figura 4.19.

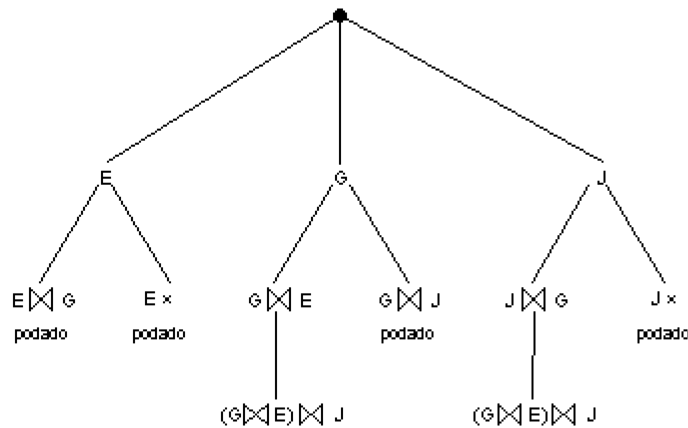



Figura 4.18. Alternativas para el ordenamiento de juntas.

Consideremos ahora el caso en donde hay más de dos relaciones en la junta. El objetivo sigue siendo transferir los operandos más pequeños. La dificultad aparece del hecho de que las operaciones de juntas pueden reducir o incrementar el tamaño de los resultados intermedios. Así, estimar el tamaño de los resultados de las juntas es obligatorio pero difícil. Una solución es estimar el costo de comunicación de todas las estrategias posibles y elegir la de costo menor. Sin embargo, el número de estrategias crece rápidamente con el número de relaciones. Este enfoque, utilizado por System R*, hace que la optimización sea costosa pero se puede amortizar si la consulta se ejecuta con frecuencia.

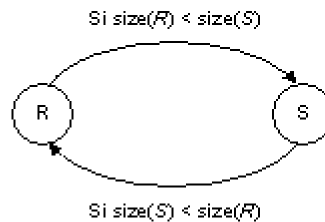



Figura 4.19. Transferencia de operandos en una operación binaria.

**Ejemplo 4.19**. Considere la siguiente consulta expresada en el álgebra relacional:

J > < $_{JNO}$ E > < $_{ENO}$ G

cuya gráfica de juntas se presenta en la Figura 4.20. Note que se han hecho suposiciones acerca de la ubicación de cada relación. Esta consulta se puede ejecutar en al menos cinco estrategias diferentes. R ® nodo *j* denota que la relación R se transfiere al nodo *j*.

1. E ® nodo 2

   Nodo 2 calcula E’ = E > < G

   E’ ® nodo 3

   Nodo 3 calcula E’ > < J

2. G ® nodo 1

   Nodo 1 calcula E’ = E > < G

   E’ ® nodo 3

   Nodo 3 calcula E’ > < J

3. G ® nodo 3

   Nodo 3 calcula G’ = G > < J

   G’ ® nodo 1

   Nodo 1 calcula G’ > < E

4. J ® nodo 2

   Nodo 2 calcula J’ = J > < G

   J’ ® nodo 1

   Nodo 1 calcula J’ > < E

5. E ® nodo 2

   J ® nodo 2

   Nodo 2 calcula E > < J > < G

Figura 4.20. Gráfica de juntas distribuida.

### 4.8.5 Optimización de consultas distribuidas

En esta sección se ilustrará brevemente el uso de las técnicas presentadas previamente en las extensiones distribuidas de INGRES y System R.

**Algoritmo Distribuido de INGRES**

El algoritmo de optimización de consultas para INGRES distribuido se deriva del algoritmo usado en INGRES centralizado. La función objetivo del algoritmo pretende minimizar la combinación del costo de comunicación junto con el tiempo de respuesta. Dado que estos criterios pueden presentar conflictos entre sí se considera una combinación pesada de ambos factores. Este

algoritmo ignora el costo de transmisión de los datos al nodo de resultados. El algoritmo también toma ventaja de la fragmentación, pero únicamente considera fragmentación horizontal. El algoritmo utiliza mensajes de tipo broadcast, de uno a todos.

**Ejemplo 4.21**. Considere la consulta

E > < G > < J

y suponga que la relación E esta fragmentada donde la ubicación de cada fragmento se muestra a continuación:

| Nodo 1 | Nodo 2 |
|---|---|
| $E_1$ | $E_2$ |
| G | J |

Existen varias estrategias posibles de ejecución. Dos de ellas son las siguientes:

1. Ejecutar la consulta completa E > < G > < J transmitiendo $E_1$ y G al nodo 2.
2. Ejecutar (E > < G) > < J transfiriendo $E_1$ > < G y G al nodo 2.

La elección entre las dos posibilidades requiere de una estimación del tamaño de los resultados intermedios. Por ejemplo, si *size*($E_1$ > < G) > size($E_1$), la estrategia 1 es mejor.

..

**Algoritmo R***

El algoritmo de optimización de consultas distribuidas R* es una extensión sustancial a las técnicas desarrolladas para el optimizador de System R. La función de costo considera el costo de procesamiento local así como el costo de comunicación. Sin embargo, no utiliza fragmentos sino relaciones completas como unidades de distribución. La compilación de la consulta es una tarea distribuida en R*, la cual es coordinada por un nodo *maestro*. El optimizador del maestro hace todas las decisiones entre nodos, tales como la selección de los nodos de ejecución así como el método de transferencia de datos. Los nodos *aprendices* que son todos los otros nodos involucrados en la consulta hacen únicamente decisiones locales, tales como, el orden de las juntas en la consulta local y generan planes de acceso local para la consulta.

Para juntar dos relaciones existen tres nodos candidatos: el nodo de la primera relación, el nodo de la segunda relación o un tercer nodo (por ejemplo, el nodo donde se hará una junta con una tercera relación). En R*, se utilizan dos métodos para las transferencias de datos entre nodos.

1. *Transfiere todo*. La relación completa es transferida al nodo donde se realiza la junta y almacenada en una relación temporal antes de hacer la junta. Si el algoritmo de la junta es por medio de una mezcla, la relación no necesita ser almacenada y el nodo que ejecuta la junta puede procesar los tuplos conforme llegan. Este método requiere de grandes transferencias de datos, pero utiliza pocos mensajes. Es adecuado si las relaciones son relativamente pequeñas.
2. *Lee conforme lo necesita*. La relación externa es recorrida secuencialmente y, para cada tuplo, el valor de la junta se envía al nodo de la relación interna, el cual selecciona los tuplos internos que satisfacen el predicado de la junta y envía los tuplos seleccionados al nodo con la relación externa. La cantidad de mensajes es proporcional al tamaño de la

relación externa. Este método es adecuado si las relaciones son grandes y la selectividad de la junta es buena.

Dada la junta de una relación externa *R* con una relación interna *S* en el atributo *A*, existen cuatro estrategias para realizar la junta. A continuación se describirá cada estrategia en detalle y se proporcionará en cada caso una versión simplificada de la fórmula de costo. *LC* denota el costo de procesamiento local (I/O + CPU) y CC denota el costo de comunicación.

**Estrategia 1**. Se transfiere la relación externa completa al nodo con la relación interna. En este caso los tuplos externos se pueden juntar con *S* conforme llegan. Así se tiene

*Costo total* = *LC*( recuperar *card*(*R*) tuplos de *R*)

+ *CC*(*size*(*R*))

+ *LC*( recuperar *s* tuplos de *S*)**card*(*R*)

**Estrategia 2**. Se transfiere la relación interna completa al nodo con la relación externa. En este caso los tuplos internos no se pueden juntar conforme llegan y se tienen que almacenar en una relación temporal *T*.

*Costo total* = *LC*( recuperar *card*(*S*) tuplos de *S*)

+ *CC*(*size*(*S*))

+ *LC*( almacenar *card*(*S*) tuplos en *T*)

+ *LC*( recuperar *card*(*R*) tuplos de *R*)

+ *LC*( recuperar *s* tuplos de *T*)**card*(*R*)

**Estrategia 3**. Se obtienen los tuplos de la relación interna conforme se necesitan para cada tuplo de la relación externa. En este caso, para cada tuplo de *R*, el atributo de junta se envía al nodo de *S*. Luego *s* tuplos de *S* que satisfacen el atributo son recuperados y envíados al nodo de *R* para ser juntados conforme llegan.

*Costo total* = *LC*( recuperar *card*(*R*) tuplos de *R*)

+ *CC*(*length*(*A*)) * *card*(*R*)

+ *LC*( recuperar *s* tuplos de *S*)**card*(*R*)

+ *CC*(*s* * *length*(*S*)) * *card*(*R*)

**Estrategia 4**. Se mueven las relaciones a un tercer nodo y se calcula la junta ahí. En este caso la relación interna se mueve al tercer nodo y se almacena en una relación temporal *T*. Luego la relación externa se mueve al tercer nodo y sus tuplos se juntan con *T* conforme llegan.

*Costo total* = *LC*( recuperar *card*(*S*) tuplos de *S*)

+ *CC*(*size*(*S*))

+ *LC*( almacenar *card*(*S*) tuplos en *T*)

+ *LC*( recuperar *card*(*R*) tuplos de *R*)

+ *CC*(*size*(*R*))

+ *LC*( recuperar *s* tuplos de *T*)**card*(*R*)

## 4.9 OPTIMIZACION LOCAL DE CONSULTAS

El último paso en el esquema de procesamiento de consultas distribuidas es la optimización de consultas locales. Después del material presentado en este capítulo, debe quedar claro que para ello se utilizan las técnicas de optimización de consultas centralizadas. El propósito es seleccionar el mejor camino de acceso para una consulta local.

# CAPITULO 5. MANEJO DE TRANSACCIONES DISTRIBUIDAS

Hasta este momento, las primitivas básicas de acceso que se han considerado son las consultas (queries). Sin embargo, no se ha discutido qué pasa cuando, por ejemplo, dos consultas tratan de actualizar el mismo elemento de datos, o si ocurre una falla del sistema durante la ejecución de una consulta. Dada la naturaleza de una consulta, de lectura o actualización, a veces no se puede simplemente reactivar la ejecución de una consulta, puesto que algunos datos pueden haber sido modificados antes la falla. El no tomar en cuenta esos factores puede conducir a que la información en la base de datos contenga datos incorrectos.

El concepto fundamental aquí es la noción de "ejecución consistente" o "procesamiento confiable" asociada con el concepto de una consulta. El concepto de una *transacción* es usado dentro del dominio de la base de datos como una unidad básica de cómputo consistente y confiable.

### 5.1 Definición de una transacción

Una ***transacción*** es una colección de acciones que hacen transformaciones consistentes de los estados de un sistema preservando la consistencia del sistema. Una base de datos está en un estado *consistente* si obedece todas las restricciones de integridad definidas sobre ella. Los cambios de estado ocurren debido a actualizaciones, inserciones, y supresiones de información. Por supuesto, se quiere asegurar que la base de datos nunca entra en un estado de inconsistencia. Sin embargo, durante la ejecución de una transacción, la base de datos puede estar temporalmente en un estado inconsistente. El punto importante aquí es asegurar que la base de datos regresa a un estado consistente al fin de la ejecución de una transacción (Ver Figura 5.1)

Lo que se persigue con el manejo de transacciones es por un lado tener una transparencia adecuada de las acciones concurrentes a una base de datos y por otro lado tener una transparencia adecuada en el manejo de las fallas que se pueden presentar en una base de datos.

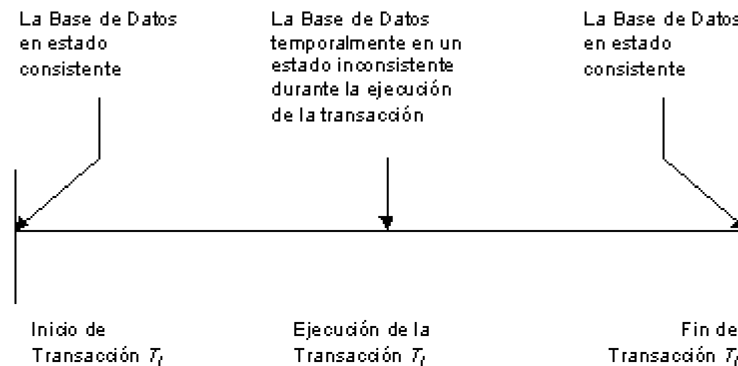



Figura 5.1. Un modelo de transacción.

**Ejemplo 5.1**. Considere la siguiente consulta en SQL para incrementar el 10% del presupuesto del proyecto CAD/CAM de la base de datos de ejemplo.

```
UPDATE J

SET BUDGET = BUDGET*1.1

WHERE JNAME = "CAD/CAM"
```

Esta consulta puede ser especificada, usando la notación de SQL, como una transacción otorgándole un nombre:

```
Begin_transaction ACTUALIZA_PRESUPUESTO
begin
        UPDATE J
        SET BUDGET = BUDGET*1.1
        WHERE JNAME = "CAD/CAM"
end.
```

**Ejemplo 5.2**. Considere una agencia de reservaciones para líneas aéreas con las siguientes relaciones:

FLIGHT( FNO, DATE, SRC, DEST, STSOLD, CAP )

CUST( CNAME, ADDR, BAL )

FC( FNO, DATE, CNAME, SPECIAL )

Una versión simplificada de una reservación típica puede ser implementada mediante la siguiente transacción:

```
Begin_transaction Reservación
begin
        input( flight_no, date, customer_name );
        EXEC SQL UPDATE FLIGHT
                SET STSOLD = STSOLD + 1
                WHERE FNO = flight_no
                AND DATE = date
        EXEC SQL INSERT
                INTO FC( FNAME, DATE, CNAME, SPECIAL )
                VALUES (flight_no, date, customer_name, null )
        output("reservación terminada");
end.
```

### Condiciones de terminación de una transacción

Una transacción siempre termina, aun en la presencia de fallas. Si una transacción termina de manera exitosa se dice que la transacción hace un *commit* (se usará el término en inglés cuando no exista un término en español que refleje con brevedad el sentido del término en inglés). Si la transacción se detiene sin terminar su tarea, se dice que la transacción *aborta*. Cuando la transacción es abortada, su ejecución es detenida y todas sus acciones ejecutadas hasta el momento son deshechas (*undone*) regresando a la base de datos al estado antes de su ejecución. A esta operación también se le conoce como *rollback*.

**Ejemplo 5.3**. Considerando de nuevo el Ejemplo 5.2, veamos el caso cuando no esisten asientos disponibles para hacer la reservación.

```
Begin_transaction Reservación
begin
        input( flight_no, date, customer_name );
        EXEC SQL SELECT STSOLD, CAP
        INTO temp1, temp2
        FROM FLIGHT
        WHERE FNO = flight_no AND DATE = date
        if temp1 = temp2 then
        output( "no hay asientos libres" )
        Abort
        else
                EXEC SQL UPDATE FLIGHT
                        SET STSOLD = STSOLD + 1
                        WHERE FNO = flight_no AND DATE = date
                EXEC SQL INSERT
                        INTO FC( FNAME, DATE, CNAME, SPECIAL )
                        VALUES (flight_no, date, customer_name, null )
                Commit
                output("reservación terminada");
        endif
end.
```

### Caracterización de transacciones

Observe que en el ejemplo anterior las transacciones leen y escriben datos. Estas acciones se utilizan como base para caracterizar a las transacciones. Los elementos de datos que lee una transacción se dice que constituyen el *conjunto de lectura* (*RS*). Similarmente, los elementos de datos que una transacción escribe se les denomina el *conjunto de escritura* (*WS*). Note que los conjuntos de lectura y escritura no tienen que ser necesariamente disjuntos. La unión de ambos conjuntos se le conoce como el *conjunto base* de la transacción ($BS = RS \cup WS$).

**Ejemplo 5.4**. Los conjuntos de lectura y escritura de la transacción del Ejemplo 5.3 son:

*RS*[Reservación] = { FLIGHT.STSOLD, FLIGHT.CAP }

*WS*[Reservación] = { FLIGHT.STSOLD, FC.FNO, FC.DATE, FC.NAME, FC.SPECIAL }

¨

### Formalización del concepto de transacción

Sea $O_{ij}(x)$ una operación $O_j$ de la transacción $T_i$ la cual trabaja sobre alguna entidad $x$. $O_j \in$ {read, write} y $O_j$ es una operación atómica, esto es, se ejecuta como una unidad indivisible. Se denota por $OS_i = \cup_j O_{ij}$ al conjunto de todas las operaciones de la transacción $T_i$. También, se denota por $N_i$ la condición de terminación para $T_i$, donde, $N_i \in$ {abort, commit}.

La transacción $T_i$ es un orden parcial, $T_i = \{ \Sigma_i, <_i \}$, donde

1. $\Sigma_i = OS_i \cup \{N_i\}$
2. Para cualesquiera dos operaciones, $O_{ij}, O_{ik} \in OS_i$, si $O_{ij} = R(x)$ y $O_{ik} = W(x)$ para cualquier elemento de datos $x$, entonces, ó $O_{ij} <_i O_{ik}$ ó $O_{ik} <_i O_{ij}$
3. $\forall O_{ij} \in OS_i, O_{ij} <_i N_i$

**Ejemplo 5.5**. Considere una transacción simple T que consiste de los siguientes pasos:

Read( $x$ )

Read( $y$ )

$x \leftarrow x + y$

Write( $x$ )

Commit

La especificación de su transacción de acuerdo con la notación formal que se ha introducido es la siguiente:

$\Sigma = \{ R(x), R(y), W(x), C \}$

$<_i = \{ (R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C) \}$

¨

Note que la relación de ordenamiento especifica el orden relativo de todas las operaciones con respecto a la condición de terminación. Esto se debe a la tercera condición de la definición de transacción. También note que no se define el ordenamiento entre cualquier par de operaciones, esto es, debido a que se ha definido un orden parcial.

**Propiedades de las transacciones**

La discusión en la sección previa clarifica el concepto de transacción. Sin embargo, aun no se ha dado ninguna justificación para afirmar que las transacciones son unidades de procesamiento consistentes y confiables. Las propiedades de una transacción son las siguientes:

1. ***Atomicidad***. Se refiere al hecho de que una transacción se trata como una unidad de operación. Por lo tanto, o todas las acciones de la transacción se realizan o ninguna de ellas se lleva a cabo. La atomicidad requiere que si una transacción se interrumpe por una falla, sus resultados parciales deben ser deshechos. La actividad referente a preservar la atomicidad de transacciones en presencia de abortos debido a errores de entrada, sobrecarga del sistema o interbloqueos se le llama *recuperación de transacciones*. La actividad de asegurar la atomicidad en presencia de caídas del sistema se le llama *recuperación de caídas*.
2. ***Consistencia***. La consistencia de una transacción es simplemente su correctitud. En otras palabras, una transacción es un programa correcto que lleva la base de datos de un estado consistente a otro con la misma característica. Debido a esto, las transacciones no violan las restricciones de integridad de una base de datos.
3. ***Aislamiento***. Una transacción en ejecución no puede revelar sus resultados a otras transacciones concurrentes antes de su *commit*. Más aún, si varias transacciones se ejecutan concurrentemente, los resultados deben ser los mismos que si ellas se hubieran ejecutado de manera secuencial (seriabilidad).
4. ***Durabilidad***. Es la propiedad de las transacciones que asegura que una vez que una transacción hace su *commit*, sus resultados son permanentes y no pueden ser borrados de la base de datos. Por lo tanto, los DBMS aseguran que los resultados de una transacción sobrevivirán a fallas del sistema. Esta propiedad motiva el aspecto de *recuperación de bases de datos*, el cual trata sobre como recuperar la base de datos a un estado consistente en donde todas las acciones que han hecho un commit queden reflejadas.

En resumen, las transacciones proporcionan una ejecución atómica y confiable en presencia de fallas, una ejecución correcta en presencia de accesos de usuario múltiples y un manejo correcto de réplicas (en el caso de que se soporten).

**Tipos de Transacciones**

Las transacciones pueden pertenecer a varias clases. Aun cuando los problemas fundamentales son los mismos para las diferentes clases, los algoritmos y técnicas que se usan para tratarlas pueden ser considerablemente diferentes. Las transacciones pueden ser agrupadas a lo largo de las siguientes dimensiones:

1. **Areas de aplicación**. En primer lugar, las transacciones se pueden ejecutar en aplicaciones no distribuidas. Las transacciones que operan en datos distribuidos se les conoce como transacciones distribuidas. Por otro lado, dado que los resultados de una transacción que realiza un commit son durables, la única forma de deshacer los efectos de una transacción con commit es mediante otra transacción. A este tipo de transacciones se les conoce como transacciones *compensatorias*. Finalmente, en ambientes heterogéneos se presentan transacciones *heterogéneas* sobre los datos.

2. **Tiempo de duración**. Tomando en cuenta el tiempo que transcurre desde que se inicia una transacción hasta que se realiza un commit o se aborta, las transacciones pueden ser de tipo batch o en línea. Estas se pueden diferencias también como transacciones de corta y larga vida. Las transacciones en línea se caracterizan por tiempos de respuesta muy cortos y por accesar un porción relativamente pequeña de la base de datos. Por otro lado, las transacciones de tipo batch toman tiempos relativamente largos y accesan grandes porciones de la base de datos.

3. **Estructura**. Considerando la estructura que puede tener una transacción se examinan dos aspectos: si una transacción puede contener a su vez subtransacciones o el orden de las acciones de lectura y escritura dentro de una transacción.

**Estructura de transacciones**

Las transacciones planas consisten de una secuencia de operaciones primitivas encerradas entre las palabras clave **begin** y **end**. Por ejemplo,

> **Begin_transaction** Reservación
>
> . . .
>
> **end**.

En las transacciones anidadas las operaciones de una transacción pueden ser así mismo transacciones. Por ejemplo,

> **Begin_transaction** Reservación
>
> > . . .
> >
> > **Begin_transaction** Vuelo
> >
> > . . .
> >
> > **end**. {Vuelo}
> >
> > . . .
> >
> > **Begin_transaction** Hotel
> >
> > . . .
> >
> > **end**.
> >
> > . . .
>
> **end**.

Una transacción anidada dentro de otra transacción conserva las mismas propiedades que la de sus padres, esto implica, que puede contener así mismo transacciones dentro de ella. Existen restricciones obvias en una transacción anidada: debe empezar *después* que su padre y debe terminar *antes* que él.

Más aún, el commit de una subtransacción es condicional al commit de su padre, en otras palabras, si el padre de una o varias transacciones aborta, las subtransacciones hijas también serán abortadas.

Las transacciones anidadas proporcionan un nivel más alto de concurrencia entre transacciones. Ya que una transacción consiste de varios transacciones, es posible tener más concurrencia dentro de una sola transacción. Así también, es posible recuperarse de fallas de manera independiente de cada subtransacción. Esto limita el daño a un parte más pequeña de la transacción, haciendo que costo de la recuperación sea menor.

En el segundo punto de vista se considera el orden de las lecturas y escrituras. Si las acciones de lectura y escritura pueden ser mezcladas sin ninguna restricción, entonces, a este tipo de transacciones se les conoce como *generales*. En contraste, si se restringe o impone que un dato deber ser leído antes de que pueda ser escrito entonces se tendrán transacciones *restringidas*. Si las transacciones son restringidas a que todas las acciones de lectura se realicen antes de las acciones de escritura entonces se les conoce como de *dos pasos*. Finalmente, existe un modelo de *acción* para transacciones restringidas en donde se aplica aún más la restricción de que cada par <read,write> tiene que ser ejecutado de manera atómica.

**Ejemplo 5.6**. Los siguientes son algunos ejemplos de los modelos de transacción mencionados antes.

General: $T_1$: { $R(x)$, $R(y)$, $W(y)$, $R(z)$, $W(x)$, $W(z)$, $W(w)$, $C$}

Dos pasos: $T_2$: { $R(x)$, $R(y)$, $R(z)$, $W(x)$, $W(y)$, $W(z)$, $W(w)$, $C$}

Restringida: $T_3$: { $R(x)$, $R(y)$, $W(y)$, $R(z)$, $W(x)$, $W(z)$, $R(w)$, $W(w)$, $C$}

Restringida y de dos pasos:

$T_4$: { $R(x)$, $R(y)$, $R(z)$, $R(w)$, $W(y)$, $W(x)$, $W(z)$, $W(w)$, $C$}

Acción: $T_5$: { [$R(x)$, $W(x)$], [$R(y)$, $W(y)$], [$R(z)$, $W(z)$], [$R(w)$, $W(w)$], $C$}

note que cada par de acciones encerrado en [ ] se ejecuta de manera atómica

..

**Aspectos relacionados al procesamiento de transacciones**

Los siguientes son los aspectos más importantes relacionados con el procesamiento de transacciones:

1. Modelo de estructura de transacciones. Es importante considerar si las transacciones son planas o pueden estar anidadas.
2. Consistencia de la base de datos interna. Los algoritmos de control de datos semántico tienen que satisfacer siempre las restricciones de integridad cuando una transacción pretende hacer un commit.
3. Protocolos de confiabilidad. En transacciones distribuidas es necesario introducir medios de comunicación entre los diferentes nodos de una red para garantizar la atomicidad y durabilidad de las transacciones. Así también, se requieren protocolos para la recuperación local y para efectuar los compromisos (commit) globales.

4. Algoritmos de control de concurrencia. Los algoritmos de control de concurrencia deben sincronizar la ejecución de transacciones concurrentes bajo el criterio de correctitud. La consistencia entre transacciones se garantiza mediante el aislamiento de las mismas.
5. Protocolos de control de réplicas. El control de réplicas se refiere a cómo garantizar la consistencia mutua de datos replicados. Por ejemplo se puede seguir la estrategia read-one-write-all (ROWA).

**Incorporación del manejador de transacciones a la arquitectura de un SMBD**

Con la introducción del concepto de transacción, se requiere revisar el modelo arquitectural presentado en el capítulo 2. En esta sección, se expande el papel del monitor de ejecución distribuida.

El monitor de ejecución distribuida consiste de dos módulos: El *administrador de transacciones* (TM) y el *despachador* (SC). Como se puede apreciar en la Figura 5.2, el administrador de transacciones es responsable de coordinar la ejecución en la base de datos de las operaciones que realiza una aplicación. El despachador, por otra parte, es responsable de implementar un algoritmo específico de control de concurrencia para sincronizar los accesos a la base de datos.

Un tercer componente que participa en el manejo de transacciones distribuidas es el *administrador de recuperación local* cuya función es implementar procedimientos locales que le permitan a una base de datos local recuperarse a un estado consistente después de una falla.

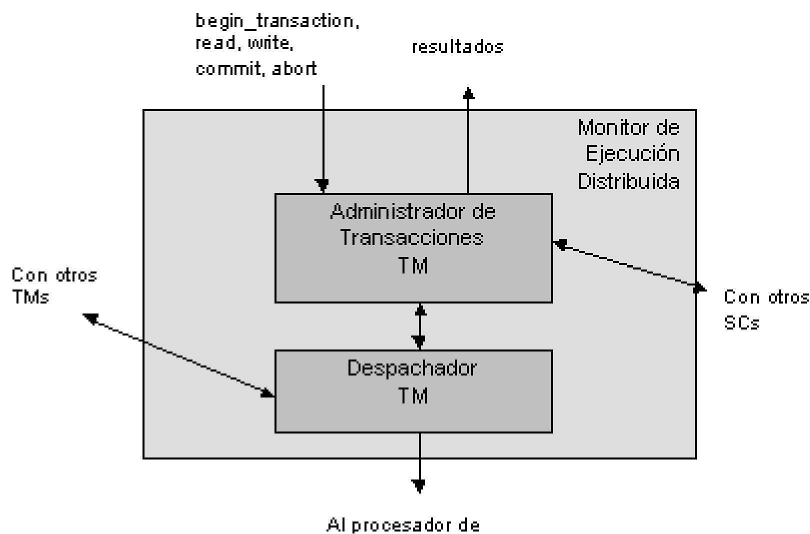



Figura 5.2. Un modelo del administrador de transacciones.

Los administradores de transacciones implementan una interfaz para los programas de aplicación que consiste de los comandos:

1. **Begin_transaction**.
2. **Read**.
3. **Write**.
4. **Commit**.
5. **Abort**.

En la Figura 5.3 se presenta la arquitectura requerida para la ejecución centralizada de transacciones. Las modificaciones requeridas en la arquitectura para una ejecución distribuida se pueden apreciar en

las Figura 5.4. En esta última figura se presentan también los protocolos de comunicación necesarios para el manejo de transacciones distribuidas.

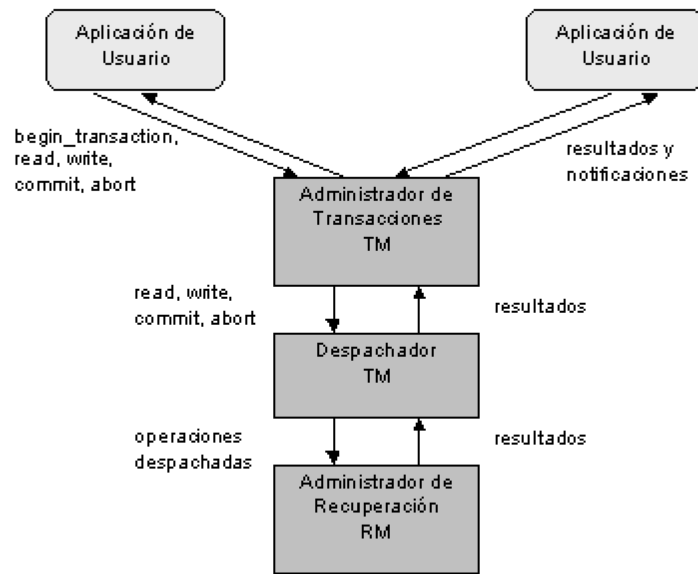



Figura 5.3. Ejecución centralizada de transacciones.

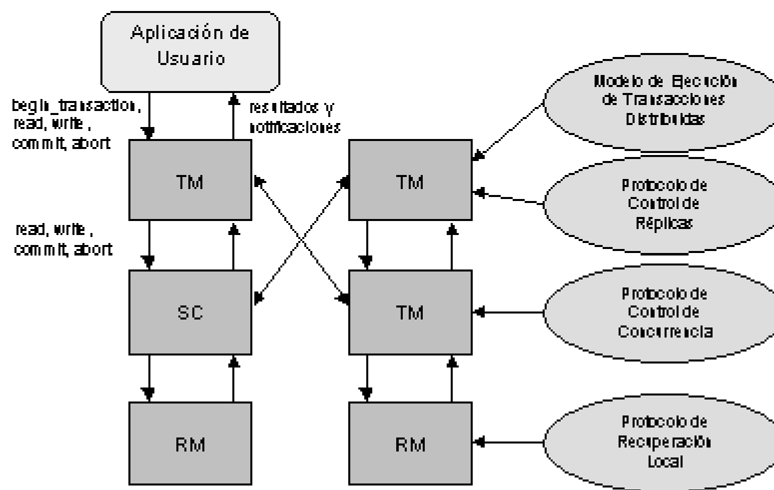



Figura 5.4. Ejecución distribuida de transacciones.

## CONTROL DE CONCURRENCIA

El control de concurrencia trata con los problemas de aislamiento y consistencia del procesamiento de transacciones. El control de concurrencia distribuido de una DDBMS asegura que la consistencia de la base de datos se mantiene en un ambiente distribuido multiusuario. Si las transacciones son internamente consistentes, la manera más simple de lograr este objetivo es ejecutar cada transacción sola, una después de otra. Sin embargo, esto puede afectar grandemente el desempeño de un DDBMS dado que el nivel de concurrencia se reduce al mínimo. El nivel de concurrencia, el número de transacciones activas, es probablemente el parámetro más importante en sistemas distribuidos. Por lo tanto, los mecanismos de control de concurrencia buscan encontrar un balance entre el mantenimiento de la consistencia de la base de datos y el mantenimiento de un alto nivel de concurrencia.

Si no se hace un adecuado control de concurrencia, se pueden presentar dos anomalías. En primer lugar, se pueden perder actualizaciones provocando que los efectos de algunas transacciones no se reflejen en la base de datos. En segundo término, pueden presentarse recuperaciones de información inconsistentes.

En este capítulo se hace la suposición de que el sistema distribuido es completamente confiable y no experimente falla alguna. En el capítulo siguiente, se considerará la presencia de fallas para obtener sistemas confiables.

## 6.1 Teoría de la seriabilidad

Una *calendarización* (*schedule*), también llamado una *historia*, se define sobre un conjunto de transacciones $T = \{ T_1, T_2, ..., T_n \}$ y especifica un orden entrelazado de la ejecución de las operaciones de las transacciones. La calendarización puede ser especificada como un orden parcial sobre $T$.

**Ejemplo 6.1**. Considere las siguientes tres transacciones:

| $T_1$: Read( $x$ ) | $T_2$: Write( $x$ ) | $T_3$: Read( $x$ ) |
|---|---|---|
| Write( $x$ ) | Write( $y$ ) | Read( $y$ ) |
| Commit | Read( $z$ ) | Read( $z$ ) |
| | Commit | Commit |

Una calendarización de las acciones de las tres transacciones anteriores puede ser:

$$H_1 = \{ W_2(x), R_1(x), R_3(x), W_1(x), C_1, W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3 \}$$

..

Dos operaciones $O_{ij}(x)$ y $O_{kl}(x)$ ($i$ y $k$ no necesariamente distintos) que accesan el mismo dato de la base de datos $x$ se dice que están en *conflicto* si al menos una de ellas es una escritura. De esta manera, las operaciones de lectura no tienen conflictos consigo mismas. Por tanto, existen dos tipos de conflictos *read-write* (o *write-read*) y *write-write*. Las dos operaciones en conflicto pueden pertenecer a la misma transacción o a transacciones diferentes. En el último caso, se dice que las transacciones tienen *conflicto*. De manera intuitiva, la existencia de un conflicto entre dos operaciones indica que su orden de ejecución es importante. El orden de dos operaciones de lectura es insignificante.

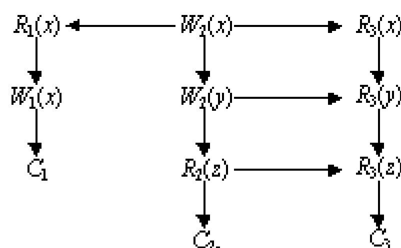
Una *calendarización completa* define el orden de ejecución de todas las operaciones en su dominio. Formalmente, una calendarización completa $S_T^c$ definido sobre un conjunto de transacciones $T = \{ T_1, T_2, ..., T_n \}$ es un orden parcial $S_T^c = \{ S_T, <_T \}$ en donde

1. S $_T$ = U ¡ S $_i$ , para todos los $i = 1, 2, ..., n$
2. $<_T$ Ê $_i$ $<_i$ , para todos los $i = 1, 2, ..., n$
3. Para cualesquiera dos operaciones en conflicto $O_{ij}$ y $O_{kl}$ Î S $_T$, ó $O_{ij} <_T O_{kl}$ ó

   $O_{kl} <_T O_{ij}$

La primera condición establece simplemente que el dominio de la calendarización es la unión de los dominios de las transacciones individuales. La segunda condición define la relación de ordenamiento como un superconjunto de la relación de ordenamiento de transacciones individuales. Esto mantiene el ordenamiento de las operaciones dentro de cada transacción. La condición final define el orden de ejecución entre dos operaciones en conflicto.

**Ejemplo 6.2**. Considere las tres transacciones del Ejemplo 6.1, una posible calendarización completa está dada por la siguiente gráfica dirigida acíclica (DAG).




Una *calendarización* se define como un prefijo de una calendarización completa. Un prefijo de un orden parcial se define como sigue. Dado un orden parcial *P* = { S , < }, *P*' = { S ', <' }, es un *prefijo* de *P* si

1. S ' Ì S *i*
2. " $e_i$ Î S ', $e_1$ <' $e_2$, si y solamente si, $e_1$ < $e_2$, y
3. " $e_i$ Î S ', si $ $e_j$ Î S y $e_j$ < $e_i$, entonces, $e_j$ Î S '

Las primeras dos condiciones definen a *P*' como una restricción de *P* en el dominio S ', en donde las relaciones de ordenamiento en *P* se mantienen por *P*'. La última condición indica que para cualquier elemento de S ', todos sus predecesores en S deben ser incluidos también en S '.

**Ejemplo 6.3**. La siguiente calendarización es un prefijo de la calendarización del Ejemplo 6.2.

Si en una calendarización *S*, las operaciones de varias transacciones no están entrelazadas, esto es, si las operaciones de una transacción ocurren de manera consecutiva, entonces se dice que la calendarización es *serial*. Si cada transacción es consistente (obedece las reglas de integridad), entonces la base de datos se garantiza ser consistente al final de la calendarización serial. La historia asociada a este tipo de calendarización se le conoce como *serial.*

**Ejemplo 6.4**. La siguiente es una historia serial para el Ejemplo 6.1.

$$H_S = \{ W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3 \}$$

Las transacciones se ejecutan de manera concurrente, pero el efecto neto de la historia resultante sobre la base de datos es *equivalente* a alguna *historia serial*. Bassada en la relación de precedencia introducida por el orden parcial, es posible discutir la equivalencia de diferentes calendarizaciones con respecto a sus efectos sobre la base de datos.

Dos calendarizaciones, $S_1$ y $S_2$, definidas sobre el mismo conjunto de transacciones *T*, se dice que son *equivalentes* si para cada par de operaciones en conflicto $O_{ij}$ y $O_{kl}$ ($i$ ¹ $k$), cada vez que $O_{ij} <_1 O_{kl}$, entonces, $O_{ij} <_2 O_{kl}$. A esta relación se le conoce como *equivalencia de conflictos* puesto que define la equivalencia de dos calendarizaciones en término del orden de ejecución relativo de las operaciones en conflicto en ellas.

Una calendarización *S* se dice que es *serializable*, si y solamente si, es equivalente por conflictos a una calendarización serial.

**Ejemplo 6.5**. Las siguientes calendarizaciones no son equivalentes por conflicto:

$H_S$ = { $W_2(x)$, $W_2(y)$, $R_2(z)$, $C_2$, $R_1(x)$, $W_1(x)$, $C_1$, $R_3(x)$, $R_3(y)$, $R_3(z)$, $C_3$ }

$H_1$ = { $W_2(x)$, $R_1(x)$, $R_3(x)$, $W_1(x)$, $C_1$, $W_2(y)$, $R_3(y)$, $R_2(z)$, $C_2$, $R_3(z)$, $C_3$ }

Las siguientes calendarizaciones son equivalentes por conflictos; por lo tanto $H_2$ es serializable:

$H_S$ = { $W_2(x)$, $W_2(y)$, $R_2(z)$, $C_2$, $R_1(x)$, $W_1(x)$, $C_1$, $R_3(x)$, $R_3(y)$, $R_3(z)$, $C_3$ }

$H_2$ = { $W_2(x)$, $R_1(x)$, $W_1(x)$, $C_1$, $R_3(x)$, $W_2(y)$, $R_3(y)$, $R_2(z)$, $C_2$, $R_3(z)$, $C_3$ }

La función primaria de un controlador de concurrencia es generar una calendarización serializable para la ejecución de todas las transacciones. El problema es, entonces, desarrollar algoritmos que garanticen que únicamente se generan calendarizaciones serializables.

## 6.2 Seriabilidad en SMBD distribuidos

En bases de datos distribuidas es necesario considerar dos tipos de historia para poder generar calendarizaciones serializables: la calendarización de la ejecución de transacciones en un nodo conocido como *calendarización local* y la *calendarización global* de las transacciones en el sistema. Para que las transacciones globales sean serializables se deben satisfacer las siguientes condiciones:

- cada historia local debe ser serializable, y
- dos operaciones en conflicto deben estar en el mismo orden relativo en todas las historias locales donde las operaciones aparecen juntas.

La segunda condición simplemente asegura que el orden de serialización sea el mismo en todos los nodos en donde las transacciones en conflicto se ejecutan juntas.

**Ejemplo 6.6**. Considere las siguientes tres transacciones:

| | |
|---|---|
| $T_1$: Read( *x* ) | $T_2$: Read( *x* ) |
| x ¬ x + 5 | x ¬ x * 5 |
| Write( *x* ) | Write( *x* ) |
| Commit | Commit |

Las siguientes historias locales son individualmente serializables (de hecho son seriales), pero las dos transacciones no son globalmente serializables.

$LH_1$ = { $R_1(x)$, $W_1(x)$, $C_1$, $R_2(x)$, $W_2(x)$, $C_2$ }

$LH_2$ = { $R_2(x)$, $W_2(x)$, $C_2$, $R_1(x)$, $W_1(x)$, $C_1$ }

## 6.3 Taxonomía de los mecanismos de control de concurrencia

El criterio de clasificación más común de los algoritmos de control de concurrencia es el tipo de primitiva de sincronización. Esto resulta en dos clases: aquellos algoritmos que están basados en acceso mutuamente exclusivo a datos compartidos (candados) y aquellos que intentar ordenar la ejecución de las transacciones de acuerdo a un conjunto de reglas (protocolos). Sin embargo, esas primitivas se pueden usar en algoritmos con dos puntos de vista diferentes: el punto de vista pesimista que considera que muchas transacciones tienen conflictos con otras, o el punto de vista optimista que supone que no se presentan muchos conflictos entre transacciones.

Los algoritmos *pesimistas* sincronican la ejecución concurrente de las transacciones en su etapa inicial de su ciclo de ejecución. Los algoritmos *optimistas* retrasan la sincronización de las transacciones hasta su terminación. El grupo de algoritmos pesimistas consiste de algoritmos *basados en candados*, algoritmos *basados en ordenamiento por estampas de tiempo* y algoritmos *híbridos*. El grupo de los algoritmos optimistas se clasifican por basados en candados y basados en estampas de tiempo (Ver. Figura 6.1).

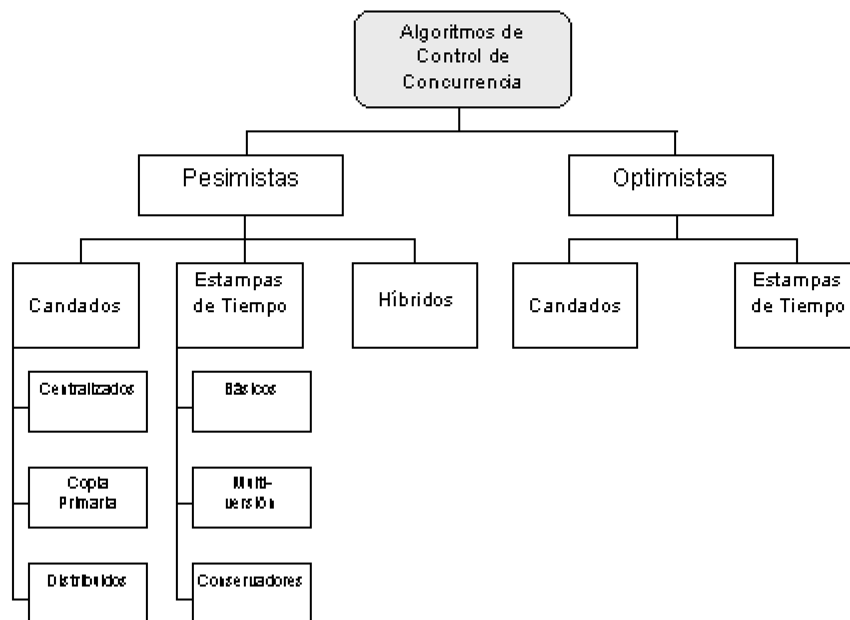



Figura 6.1. Clasificación de los algoritmos de control de concurrencia.

## 6.4 Algoritmos basados en candados

En los algoritmos basados en candados, las transacciones indican sus intenciones solicitando candados al despachador (llamado el *administrador de candados*). Los candados son de lectura (*rl*), también llamados *compartidos*, o de escritura (*wl*), también llamados *exclusivos*. Como se aprecia en la tabla siguiente, los candados de lectura presentan conflictos con los candados de escritura, dado que las operaciones de lectura y escritura son incompatibles.

| | *rl* | *wl* |
|---|---|---|
| *rl* | si | no |
| *wl* | no | no |

En sistemas basados en candados, el despachador es un *administrador de candados* (LM). El administrador de transacciones le pasa al administrador de candados la operación sobre la base de datos (lectura o escritura) e información asociada, como por ejemplo el elemento de datos que es accesado y el identificador de la transacción que está enviando la operación a la base de datos. El administrador de candados verifica si el elemento de datos que se quiere accesar ya ha sido bloqueado por un candado. Si candado solicitado es incompatible con el candado con que el dato está bloqueado, entonces, la transacción solicitante es retrasada. De otra forma, el candado se define sobre el dato en el modo deseado y la operación a la base de datos es transferida al procesador de datos. El administrador de transacciones es informado luego sobre el resultado de la operación. La terminación de una transacción libera todos los candados y se puede iniciar otra transacción que estaba esperando el acceso al mismo dato.

**Candados de dos fases (2PL)**

En los candados de dos fases una transacción le pone un candado a un objeto antes de usarlo. Cuando un objeto es bloqueado con un candado por otra transacción, la transacción solicitante debe esperar. Cuando una transacción libera un candado, ya no puede solicitar más candados. Así una transacción que utiliza candados de dos fases se comporta como en la Figura 6.2. En la primera fase solicita y adquiere todos los candados sobre los elementos que va a utilizar y en la segunda fase libera los candados obtenidos uno por uno.

La importancia de los candados de dos fases es que se ha demostrado de manera teórica que todos las calendarizaciones generadas por algoritmos de control de concurrencia que obedecen a los candados de dos fases son serializables.

Puede suceder que si una transacción aborta después de liberar un candado, otras transacciones que hayan accesado el mismo elemento de datos aborten también provocando lo que se conoce como *abortos en cascada*. Para evitar lo anterior, los despachadores para candados de dos fases implementan lo que se conoce como los *candados estrictos de dos fases* en los cuales se liberan todos los candados juntos cuando la transacción termina (con commit o aborta). El comportamiento anterior se muestra en la Figura 6.3.

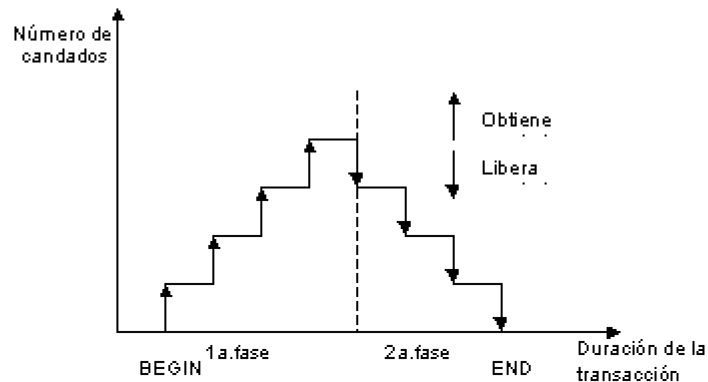



Figura 6.2. Gráfica del uso de los candados de dos fases.

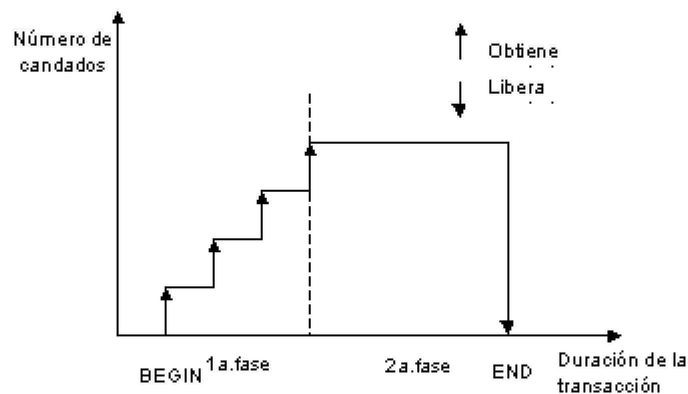



Figura 6.3. Comportamiento de los candados de dos fases estrictos.

6.4.1 Candados de dos fases centralizados

En sistemas distribuidos puede que la administración de los candados se dedique a un solo nodo del sistema, por lo tanto, se tiene un despachador central el cual recibe todas las solicitudes de candados del sistema. En la Figura 6.4 se presenta la estructura de la comunicación de un administrador centralizado de candados de dos fases. La comunicación se presenta entre el administrador de transacciones del nodo en donde se origina la transacción (llamado el *coordinador* TM), el administrador de candados en el nodo central y los procesadores de datos (DP) de todos los nodos participantes. Los nodos participantes

son todos aquellos en donde la operación se va a llevar a cabo.

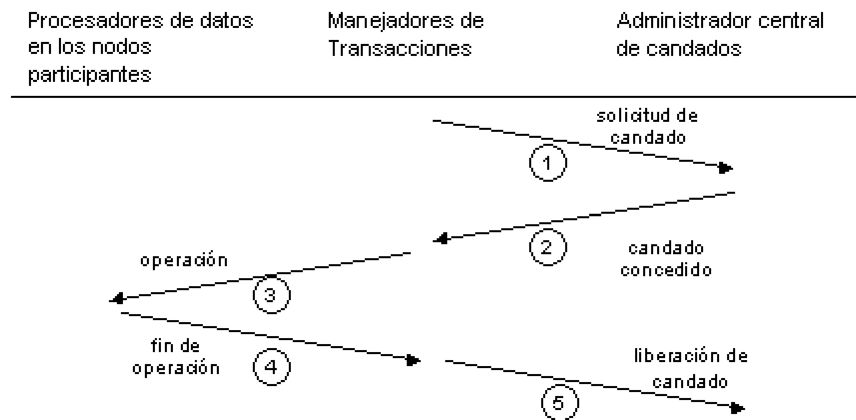



Figura 6.4. Comunicación en un administrador centralizado de candados de dos fases estrictos.

La crítica más fuerte a los algoritmos centralizados es el "cuello de botella" que se forma alrededor del nodo central reduciendo los tiempos de respuesta de todo el sistema. Más aún, su disponibilidad es reducida a cero cuando se presentan fallas en el nodo central.

### 6.4.2 Candados de dos fases distribuidos

En los candados de dos fases distribuidos se presentan despachadores en cada nodo del sistema. Cada despachador maneja las solicitudes de candados para los datos en ese nodo. Una transacción puede leer cualquiera de las copias replicada del elemento *x*, obteniendo un candado de lectura en cualquiera de las copias de *x*. La escritura sobre *x* requiere que se obtengan candados para todas las copias de *x*. La comunicación entre los nodos que cooperan para ejecutar una transacción de acuerdo al protocolo de candados distribuidos de dos fases se presenta en la Figura 6.5. Los mensajes de solicitud de candados se envían a todos los administradores de candados que participan en el sistema. Las operaciones son pasadas a los procesadores de datos por los administradores de candados. Los procesadores de datos envía su mensaje de "fin de operación" al administrador de transacciones coordinador.

## 6.5 Algoritmos basados en estampas de tiempo

A diferencia de los algoritmos basados en candados, los algoritmos basados en estampas de tiempo no pretenden mantener la seriabilidad por exclusión mutua. En lugar de eso, ellos seleccionan un orden de serialización a priori y ejecutan las transacciones de acuerdo a ellas. Para establecer este ordenamiento, el administrador de transacciones le asigna a cada transacción $T_i$ una estampa de tiempo única $ts(T_i)$ cuando ésta inicia. Una *estampa de tiempo* es un identificador simple que sirve para identificar cada transacción de manera única. Otra propiedad de las estampas de tiempo es la *monoticidad*, esto es, dos estampas de tiempo generadas por el mismo administrador de transacciones deben ser monotonicamente crecientes. Así, las estampas de tiempo son valores derivados de un dominio totalmente ordenado.

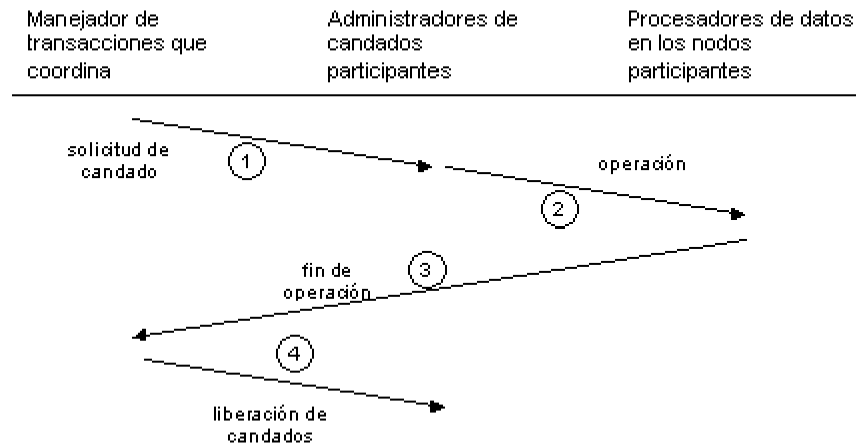



Figura 6.5. Comunicación en candados de dos fases distribuidos.

Existen varias formas en que las estampas de tiempo se pueden asignar. Un método es usar un contador global monotonicamente creciente. Sin embargo, el mantenimiento de contadores globales es un problema en sistemas distribuidos. Por lo tanto, es preferible que cada nodo asigne de manera autónoma las estampas de tiempos basándose en un contador local. Para obtener la unicidad, cada nodo le agrega al contador su propio identificador. Así, la estampa de tiempo es un par de la forma

<*contador local*, *identificador de nodo*>

Note que el identificador de nodo se agrega en la posición menos significativa, de manera que, éste sirve solo en el caso en que dos nodos diferentes le asignen el mismo contador local a dos transacciones diferentes.

El administrador de transacciones asigna también una estampa de tiempo a todas las operaciones solicitadas por una transacción. Más aún, a cada elemento de datos $x$ se le asigna una *estampa de tiempo de escritura*, $wts(x)$, y una *estampa de tiempo de lectura*, $rts(x)$; sus valores indican la estampa de tiempo más grande para cualquier lectura y escritura de $x$, respectivamente.

El ordenamiento de estampas de tiempo (TO) se realiza mediante la siguiente regla:

> **Regla TO**: dadas dos operaciones en conflicto, $O_{ij}$ y $O_{kl}$, perteneciendo a las transacciones $T_i$ y $T_k$, respectivamente, $O_{ij}$ es ejecutada antes de $O_{kl}$, si y solamente si, $ts(T_i) < ts(T_k)$. En este caso $T_i$ se dice ser un transacción *más vieja* y $T_k$ se dice ser una transacción *más joven*.

Dado este orden, un conflicto entre operaciones se puede resolver de la siguiente forma:

| | |
|---|---|
| ***for*** $R_i(x)$ ***do begin*** | ***for*** $W_i(x)$ ***do begin*** |
| if $ts(T_i) < wts(x)$ ***then*** | ***if*** $ts(T_i) < rts(x)$ ***and*** |
| *reject* $R_i(x)$ | $ts(T_i) < wts(x)$ ***then*** |
| ***else*** | *reject* $W_i(x)$ |
| *accept* $R_i(x)$ | ***else*** |

| | |
|---|---|
| $rts(x) \neg ts(T_i)$ | *accept* $W_i(x)$ |
| ***end*** | $wts(x) \neg ts(T_i)$ |
| | ***end*** |

La acción de rechazar una operación, significa que la transacción que la envió necesita reiniciarse para obtener la estampa de tiempo más reciente del dato e intentar hacer nuevamente la operación sobre el dato.

**Ordenamiento conservador por estampas de tiempo**

El ordenamiento básico por estampas de tiempo trata de ejecutar una operación tan pronto como se recibe una operación. Así, la ejecución de las operaciones es progresiva pero pueden presentar muchos reinicios de transacciones. El ordenamiento conservador de estampas de tiempo retrasa cada operación hasta que exista la seguridad de que no será reiniciada. La forma de asegurar lo anterior es sabiendo que ninguna otra operación con una estampa de tiempo menor puede llegar al despachador. Un problema que se puede presentar al retrasar las operaciones es que ésto puede inducir la creación de interbloqueos (*deadlocks*).

**Ordenamiento por estampas de tiempo múltiples**

Para prevenir la formación de interbloqueos se puede seguir la estrategia siguiente. Al hacer una operación de escritura, no se modifican los valores actuales sino se crean nuevos valores. Así, puede haber copias múltiples de un dato. Para crear copias únicas se siguen las siguientes estrategias de acuerdo al tipo de operación de que se trate:

1. Una operación de lectura $R_i(x)$ se traduce a una operación de lectura de $x$ de una sola versión encontrando la versión de $x$, digamos $x_v$, tal que, $ts(x_v)$ es la estampa de tiempo más grande que tiene un valor menor a $ts(T_i)$.
2. Una operación de escritura $W_i(x)$ se traduce en una sola version, $W_i(x_w)$, y es aceptada si el despachador no ha procesado cualquier lectura $R_j(x_r)$, tal que,

$$ts(T_i) < ts(x_r) < ts(T_j)$$

6.6 Control de concurrencia optimista

Los algoritmos de control de concurrencia discutidos antes son por naturaleza pesimistas. En otras palabras, ellos asumen que los conflictos entre transacciones son muy frecuentes y no permiten el acceso a un dato si existe una transacción conflictiva que accesa el mismo dato. Así, la ejecución de cualquier operación de una transacción sigue la secuencia de fases: validación (V), lectura (R), cómputo (C) y escritura (W) (ver Figura 6.6a). Los algoritmos optimistas, por otra parte, retrasan la fase de validación justo antes de la fase de escritura (ver Figura 6.6b). De esta manera, una operación sometida a un despachador optimista nunca es retrasada.

Las operaciones de lectura, cómputo y escrita de cada transacción se procesan libremente sin actualizar la base de datos corriente. Cada transacción inicialmente hace sus cambios en copias locales de los datos. La fase de validación consiste en verificar si esas actualizaciones conservan la consistencia de la base de datos. Si la respuesta es positiva, los cambios se hacen globales (escritos en la base de datos corriente). De otra manera, la transacción es abortada y tiene que reiniciar

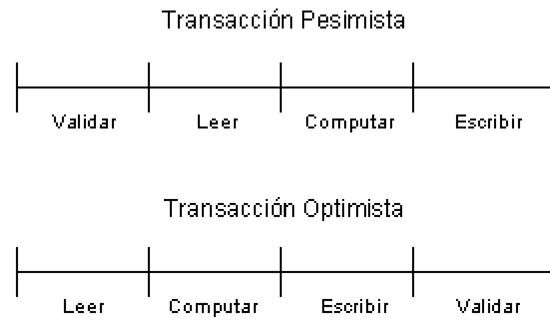



Figura 6.6. Fases de la ejecución de una transacción a) pesimista, b) optimista.

Los mecanismos optimistas para control de concurrencia fueron propuestos originalmente con el uso de estampas de tiempo. Sin embargo, en este tipo de mecanismos las estampas de tiempo se asocian únicamente con las transacciones, no con los datos. Más aún, las estampas de tiempo no se asignan al inicio de una transacción sino justamente al inicio de su fase de validación. Esto se debe a que las estampas se requieren únicamente durante la fase de validación.

Cada transacción $T_i$ se subdivide en varias subtransacciones, cada una de las cuales se puede ejecutar en nodos diferentes. Sea $T_{ij}$ una subtransacción de $T_i$ que se ejecuta en el nodo *j*. Supongamos que las transacciones se ejecutan de manera independiente y ellas alcanzan el fin de sus fases de lectura. A todas las subtransacciones se les asigna una estampa de tiempo al final de su fase de lectura. Durante la fase de validación se realiza una prueba de validación, si una transacción falla, todas las transacciones se rechazan.

La prueba de validación se realiza con una de las siguientes reglas:

1. Si todas las transacciones $T_k$, tales que, $ts( T_k ) < ts( T_{ij} )$, han terminado su fase de escritura antes que $T_{ij}$ ha iniciado su fase de lectura entonces la validación tiene éxito. En este caso la ejecución de las transacciones es completamente serial como se muestra en la Figura 6.7a.
2. Si existe alguna transacción $T_k$, tal que, $ts( T_k ) < ts( T_{ij} )$ y la cual completa su fase de escritura mientras $T_{ij}$ está en su fase de lectura, entonces, la validación tiene éxito si *WS*($T_k$ ) Ç *RS*($T_{ij}$ ) = Æ . En este caso, las fases de lectura y escritura se traslapan, como se muestra en la Figura 6.7b, pero $T_{ij}$ no lee datos queson escritos por $T_k$.
3. Si existe alguna transacción $T_k$, tal que, $ts( T_k ) < ts( T_{ij} )$ y la cual completa su fase de lectura antes que $T_{ij}$ termine su fase de lectura, entonces, la validación tiene éxito si *WS*($T_k$ ) Ç *RS*($T_{ij}$ ) = Æ y *WS*($T_k$ ) Ç *WS*($T_{ij}$ ) = Æ . En este caso, las fases de lectura se traslapan, como se muestra en la Figura 6.7c, pero las transacciones no accesan datos comunes.

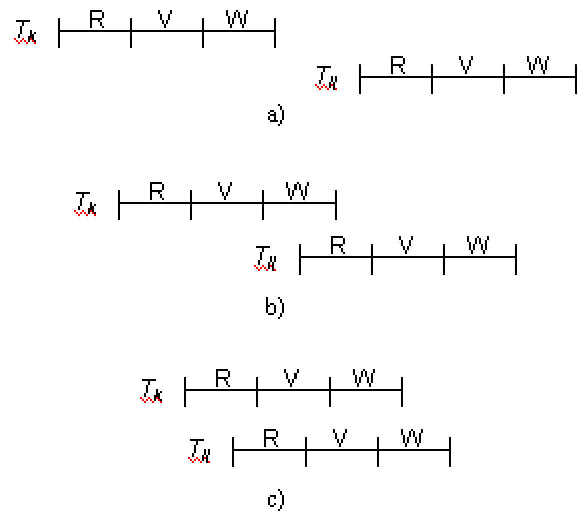



Figura 6.7. Casos diferentes de las pruebas de validación para control de concurrencia optimista

# CAPITULO 7. CONFIABILIDAD

Un sistema de manejo de bases de datos confiable es aquel que puede continua procesando las solicitudes de usuario aún cuando el sistema sobre el que opera no es confiable. En otras palabras, aun cuando los componentes de un sistema distribuido fallen, un DDMBS confiable debe seguir ejecutando las solicitudes de usuario sin violar la consistencia de la base de datos. En este capítulo se discutirán las características de un DDBMS confiable. En el capítulo 5 se vio que la confiabilidad de un DDBMS se refiere a la atomicidad y durabilidad de las transacciones. En el capítulo 6 se asumió que el sistema sobre el cual se ejecutan los mecanismos de control de concurrencia es confiable. En este capítulo se considerará el caso de que un sistema distribuido no sea confiable y, particularmente desde el punto de vista de los DDMBS, se presentarán los protocolos para recuperación de información.

## 7.1 Definiciones

A lo largo de estas notas nos hemos referido a la confiabilidad y disponibilidad de la base de datos sin definir esos términos de manera precisa. En esta sección daremos sus definiciones generales para posteriormente elaborarlas de manera más formal. La confiabilidad se puede interpretar de varias formas. La confiabilidad se puede ver como una medida con la cual un sistema conforma su comportamiento a alguna especificación. También se puede interpretar como la probabilidad de que un sistema no haya experimentado ninguna falla dentro de un periodo de tiempo dado. La confiabilidad se utiliza típicamente como un criterio para describir sistemas que no pueden ser reparados o donde la operación continua del sistema es crítica.

Disponibilidad, por otro lado, es la fracción del tiempo que un sistema satisface su especificación. En otras palabras, la probabilidad de que el sistema sea operacional en un instante dado de tiempo.

### 7.1.1 Sistema, estado y falla

Un *sistema* se refiere a un mecanismo que consiste de una colección de componentes y sus interacciones con el medio ambiente que responden a estímulos que provienen del mismo con un patrón de comportamiento reconocible (ver Figura 7.1). Cada componente de un sistema puede ser así mismo un sistema, llamado comúnmente *subsistema*. Un *estado externo* de un sistema se puede definir como la respuesta que un sistema proporciona a un estímulo externo. Por lo tanto, es posible hablar de un sistema que se mueve dentro de estados externos de acuerdo a un estímulo proveniente del medio ambiente. Un *estado interno* es, por lo tanto, la respuesta del sistema a un estímulo interno. Desde el punto de vista de confiabilidad, es conveniente definir a un estado interno como la unión de todos los estado externos de las componentes que constituyen el sistema. Así, el cambio de estado interno se da como respuesta a los estímulos del medio ambiente.

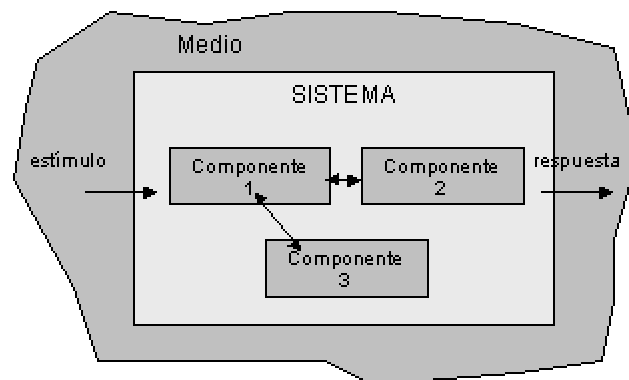



Figura 7.1. Conceptos básicos de un sistema.

El comportamiento del sistema al responder a cualquier estímulo del medio ambiente necesita establecerse por medio de una *especificación*, la cual indica el comportamiento válido de cada estado del sistema. Su especificación es no sólo necesaria para un buen diseño sino también es esencial para definir los siguientes conceptos de confiabilidad.

Cualquier desviación de un sistema del comportamiento descrito en su especificación se considera como una *falla*. Cada falla necesita ser rastreada hasta su causa. En un sistema confiable los cambios van de estados válidos a estados válidos. Sin embargo, en un sistema no confiable, es posible que el sistema caiga en un estado interno el cual no obedece a su especificación; a este tipo de estados se les conoce como *estados erróneos*. Transiciones a partir de este estado pueden causar una falla. La parte del estado interno que es incorrecta se le conoce como *error* del sistema. Cualquier error en los estados internos de las componentes del sistema se le conoce como una *falta* en el sistema. Así, una falta causa un error lo que puede inducir una falla del sistema (ver Figura 7.2).

Las faltas del sistema se pueden clasificar como severas (hard) y no severas (soft). Las faltas severas casi siempre son de tipo permanente y conducen a fallas del sistema severas. Las faltas no severas por lo general son transitorias o intermitentes. Ellas inducir fallas del sistema no severas las cuales representan, por lo general, el 90 % de todas las fallas.

Figura 7.2. De faltas a fallas

Se presentará ahora la definición formal de confiabilidad y disponibilidad. La confiabilidad de un sistema, $R(t)$, se define como la siguiente probabilidad condicional:

$$R(t) = \Pr\{\text{ 0 fallas en el tiempo } [0,t] \mid \text{no hubo fallas en } t = 0 \}$$

Si la ocurrencia de fallas sigue una distribución de Poisson, entonces,

$$R(t) = \Pr\{\text{ 0 fallas en el tiempo } [0,t] \}$$

Es posible derivar la siguiente expresión:

$$\Pr\{\text{ 0 fallas en el tiempo } [0,t] \} = \frac{e^{-m(t)}\left[m(t)\right]^k}{k!}$$

donde $m(t)$ se le conoce como la función de riesgo la cual proporciona el rango de fallas de la componente dependiente del tiempo y se define como

$$m(t) = \int_0^t z(x)dx$$

El número promedio esperado de fallas en el tiempo $[0,t]$ puede ser calculado como

$$E\left[k\right] = \sum_{k=0}^{\infty} k\,\frac{e^{-m(t)}\left[m(t)\right]^k}{k!} = m(t)$$

y la varianza se puede calcular como

$$Var[k] = E[k^2] - (E[k])^2 = m(t)$$

así, entonces la confiabilidad de una componente simple es

$$R(t) = e^{-m(t)}$$

y la de un sistema de *n* componentes no redundantes es

$$R_{sys}(t) = \prod_{i=1}^{n} R_i(t)$$

La *disponibilidad* de acuerdo a lo mencionado antes se puede establecer como

*A*(*t*) = Pr{ sistema sea operacional en el tiempo *t* }

Un número de fallas pueden haber ocurrido antes del tiempo *t*, pero si todas ellas han sido reparadas, el sistema es disponible en tiempo *t*. Es claro que la disponibilidad se refiere a sistemas que pueden ser reparados. Si se supone que las fallas siguen una distribución de Poisson con un rango l de fallas, y además el tiempo de reparación es exponencial con un tiempo de reparación medio de 1/m , la disponibilidad de un estado estable del sistema se puede escribir como:

$$A = \lim_{t \to \infty} A(t) = \frac{\mu}{\lambda + \mu}$$

El cálculo de la confiabilidad y disponibilidad puede ser tedioso. Por lo tanto, es acostumbrado usar dos métricas de un sólo parámetro para modelar el comportamiento del sistema. Las dos métricas son el *tiempo medio entre fallas* (MTBF por sus siglas en inglés) y el *tiempo medio para reparaciones* (MTTR). El MTBF puede ser calculado ó a partir de datos empíricos ó de la función de confiabilidad como:

$$\mathrm{MTBF} = \int_0^{\infty} R(t)d(t)$$

El MTTR está relacionado al rango de reparación de la misma forma que el MTBF está relacionado al rango de fallas. Usando estas dos métricas, la disponibilidad de un estado estable de un sistema con rangos de falla y reparación exponencial se puede especificar como

$$A = \frac{\mathrm{MTBF}}{\mathrm{MTBF} + \mathrm{MTTR}}$$

### 7.2 Tipos de fallas en SMBDD

Diseñar un sistema confiable que se pueda recuperar de fallas requiere identificar los tipos de fallas con las cuales el sistema tiene que tratar. Así, los tipos de fallas que pueden ocurrir en un SMBD distribuido son:

1. **Fallas de transacciones**. Las fallas en transacciones se pueden deber a un error debido a datos de entrada incorrectos así como a la detección de un interbloqueo. La forma usual de enfrentar

las fallas en transacciones es abortarlas. Experimentalmente, se ha determinado que el 3% de las transacciones abortan de manera anormal.

2. **Fallas del sistema**. En un sistema distribuido se pueden presentar fallas en el procesador, la memoria principal o la fuente de energía de un nodo. En este tipo de fallas se asume que el contenido de la memoria principal se pierde, pero el contenido del almacenamiento secundario es seguro. Típicamente, se hace diferencia entre las fallas parciales y fallas totales del nodo. Una falla total se presenta en todos los nodos del sistema distribuido. Una falla parcial se presenta solo en algunos nodos del sistema.
3. **Fallas del medio de almacenamiento**. Se refieren a las fallas que se pueden presentar en los dispositivos de almacenamiento secundario que almacenan las bases de datos. Esas fallas se pueden presentar por errores del sistema operativo, por errores del controlador del disco, o del disco mismo.
4. **Fallas de comunicación**. Las fallas de comunicación en un sistema distribuido son frecuentes. Estas se pueden manifestar como pérdida de mensajes lo que lleva en un caso extremo a dividir la red en varias subredes separadas.

## 7.3 Protocolos locales

En esta sección se discutirán las funciones realizadas por el administrador de recuperación local (LRM) el cual debe existir en cada nodo. Esas funciones mantienen las propiedades de atomicidad y durabilidad de las transacciones locales. Ellas están relacionadas a la ejecución de los comandos que son pasados al LRM, las cuales son **begin_transaction**, **read**, **write**, **commit** y **abort**.

### 7.3.1 Consideraciones estructurales

En esta parte se hará referencia al modelo de la arquitectura de un DDBMS presentado en el capítulo 2. Se revisará particularmente la interfaz entre el LRM y el administrador del buffer de la base de datos (BM). La arquitectura correspondiente a la recuperación de errores consiste de un sistema de almacenamiento constituido por dos partes. La primera, llamada memoria principal, es un medio de almacenamiento volátil. La segunda parte, llamada almacenamiento secundario, es un medio de almacenamiento permanente el cual, en principio, no es infalible a fallas. Sin embargo, por medio de una combinación de técnicas de hardware y de software es posible garantizar un medio de almacenamiento estable, capaz de recuperarse de fallas. A todos los elementos utilizados para obtener un almacenamiento estable se les agrega el atributo "*estable*" con el propósito de reconocer que ellos han sido modificados o creados para este fin. Así tendremos una base de datos estable y operaciones de lectura y escritura estables.

En la Figura 7.3 se presenta la interfaz entre el administrador de recuperación local y el administrador de buffers de la base de datos. El administrador de buffers de la base de datos mantiene en memoria principal los datos más recientemente accesados, esto se hace con el propósito de mejorar el rendimiento. Típicamente, el buffer se divide en páginas que son del mismo tamaño que las páginas de la base de datos estable. La parte de la base de datos que se encuentra en el buffer se le conoce como *base de datos volátil*. Es importante notar que el LRM ejecuta las operaciones solicitadas por una transacción sólo en la base de datos volátil. En un tiempo posterior, la base de datos volátil es escrita a la base de datos estable.

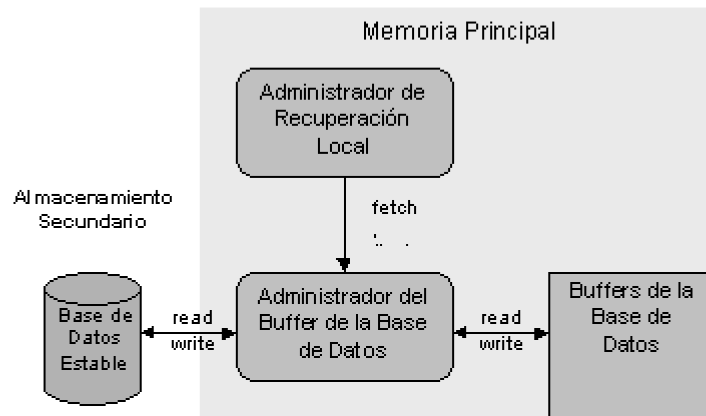



Figura 7.3. Interfaz entre el administrador local de recuperación y el administrador de buffers de la base de datos.

Cuando el LRM solicita una página de datos, envía una instrucción conocida como ***fetch***. El administrador del buffer verifica si la página ya existe en el buffer, y si es así la hace disponible a la transacción que la solicita. En caso contrario, lee la página de la base de datos estable y la coloca en un buffer vacío. Si no existe un buffer vacío, selecciona uno, la página que contiene es enviada a la base de datos estable y la página solicitada es colocada en el buffer liberado. Para forzar que se descarguen las páginas almacenadas en los buffers, existe el comando ***flush*.**

### 7.3.2 Información de recuperación

En esta sección se asumirá que ocurren únicamente fallas del sistema. Más adelante se considerará el caso de los otros tipos de fallas. Cuando una falla del sistema ocurre, el contenido de la base de datos volátil se pierde. Por lo tanto, el DBMS tiene que mantener cierta información acerca de su estado en el momento de la falla con el fin de ser capaz de llevar a la base de datos al estado en el que se encontraba antes de la falla. A esta información se le denomina *información de recuperación*.

La información de recuperación que el sistema mantiene depende del método con el que se realizan las actualizaciones. Existen dos estrategias para efectuarlas: en el lugar (*in place*) y fuera de lugar (*out-of-place*). En el primer caso, cada actualización se hace directamente en los valores almacenados en las páginas de los buffers de la base de datos. En la segunda alternativa, al crear un nuevo valor para un dato, éste se almacena en forma separada del valor anterior. De esta manera, se mantiene los valores nuevo y anterior.

### Recuperación in-place

Dado que las actualización *in-place* hacen que los valores anteriores se pierdan, es necesario mantener suficiente información de los cambios de estado en la base de datos. Esta información se mantiene, por lo general, en el *registro de la base de datos* (database log). Así cada actualización, no solo cambia la base de datos, sino es también guardada en el registro de la base de datos (Figura 7.4).

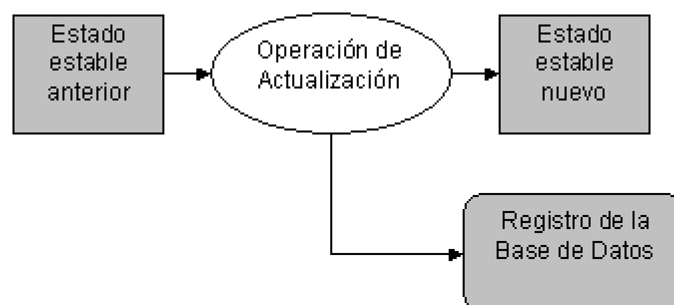


Figura 7.4. Ejecución de una operación de actualización.

El registro de la base de datos contiene información que es utilizada por el proceso de recuperación para restablecer la base de datos a un estado consistente. Esta información puede incluir entre otras cosas:

- el identificador de la transacción,
- el tipo de operación realizada,
- los datos accesados por la transacción para realizar la acción,
- el valor anterior del dato (imagen anterior), y
- el valor nuevo del dato (imagen nueva).

Considere el escenario mostrado en la Figura 7.5. El DBMS inicia la ejecución en el tiempo 0 y en el tiempo *t* se presenta una falla del sistema. Durante el periodo [0,*t*] ocurren dos transacciones, $T_1$ y $T_2$. $T_1$ ha sido concluida (ha realizado su commit) pero $T_2$ no pudo ser concluida. La propiedad de durabilidad requiere que los efectos de $T_1$ sean reflejados en la base de datos estable. De forma similar, la propiedad de atomicidad requiere que la base de datos estable no contenga alguno de los efectos de $T_2$.

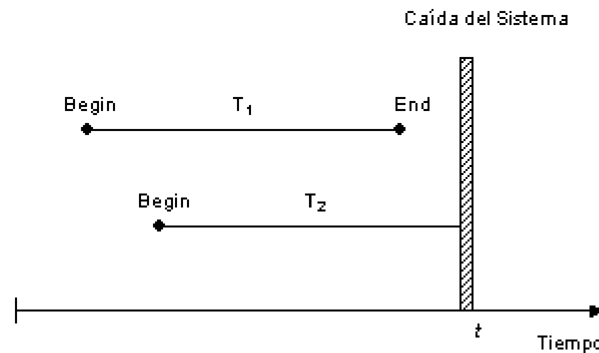



Figura 7.5. Ejemplo de una falla del sistema.

A pesar que $T_1$ haya sido terminada, puede suceder que el buffer correspondiente a la página de la base de datos modificada no haya sido escrito a la base de datos estable. Así, para este caso la recuperación tiene que volver a realizar los cambios hechos por $T_1$. A esta operación se le conoce como REDO y se presenta en la Figura 7.6. La operación de REDO utiliza la información del registro de la base de datos y realiza de nuevo las acciones que pudieron haber sido realizadas antes de la falla. La operación REDO genera una nueva imagen.

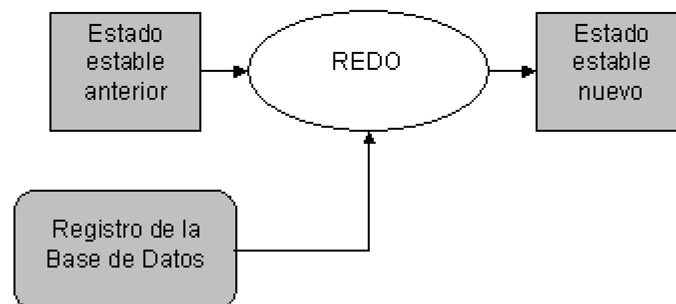



Figura 7.6. Operación REDO.

Por otra parte, es posible que el administrador del buffer haya realizado la escritura en la base de datos estable de algunas de las páginas de la base de datos volátil correspondientes a la transacción $T_2$. Así, la información de recuperación debe incluir datos suficientes para permitir deshacer ciertas actualizaciones en el nuevo estado de la base de datos y regrasarla al estado anterior. A esta operación

se le conoce como UNDO y se muestra en la Figura 7.7. La operación UNDO restablece un dato a su imagen anterior utilizando la información del registro de la base de datos.

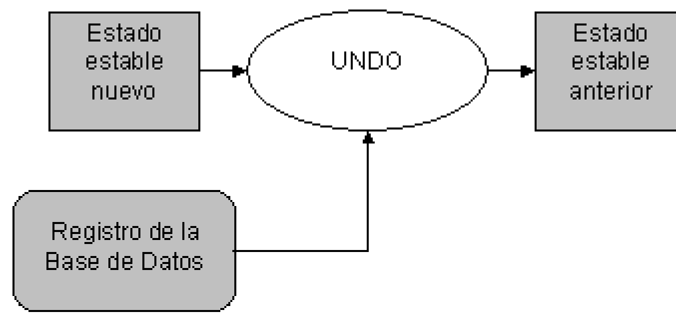



Figura 7.7. Operación UNDO.

De forma similar a la base de datos volátil, el registro de la base de datos se mantiene en memoria principal (llamada los *buffers de registro*) y se escribe al almacenamiento estable (llamado *registro estable*). Las páginas de registro se pueden escribir en el registro estable de dos formas: *síncrona* o *asíncrona*. En forma síncrona, también llamada un registro forzado, la adición de cada dato en el registro requiere que la página del registro correspondiente se mueva al almacenamiento estable. De manera asíncrona, las páginas del registro se mueven en forma periódica o cuando los buffers se llenan.

Independientemente de que la escritura del registro sea síncrona o asíncrona, se debe observar un protocolo importante para el mantenimiento del registro de la base de datos. Considere el caso donde las actualizaciones a la base de datos son escritas en el almacenamiento estable antes de que el registro sea modificado en el registro estable para reflejar la actualización. Si una falla ocurre antes de que el registro sea escrito, la base de datos permanecerá en forma actualizada, pero el registro no indicará la actualización lo que hará imposible recuperar la base de datos a un estado consistente antes de la actualización. Por lo tanto, el registro estable siempre debe ser actualizado antes de la actualización de la base de datos. A este protocolo se le conoce como registro antes de la escritura (write-ahead logging) y se puede especificar de la manera siguiente:

1. Antes de que la base de datos estable sea actualizada, las imágenes anteriores deben ser almacenadas en el registro estable. Esto facilita la realización de un UNDO.
2. Cuando la transacción realiza un commit, las imágenes nuevas tienen que ser almacenadas en el registro estable antes de la actualización de la base de datos estable. Esto facilita la realización de una operación REDO.

La interfaz completa del registro de la base de datos volátil y estable se presenta en la Figura 7.8.

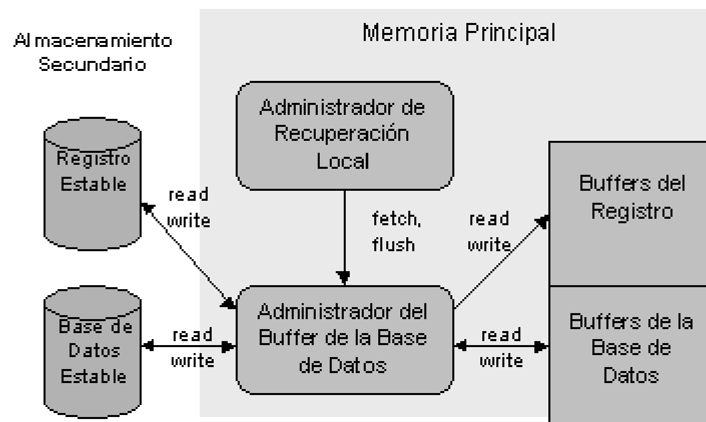


Figura 7.8. Interfaz entre el registro de la base de datos volátil y estable.

**Recuperación out-of-place**

Las técnicas de recuperación más comunes son de tipo in-place. Por lo tanto, aquí se presenta solo una breve descripción de las técnicas out-of-place.

1. **Shadowing**. Cuando ocurre una actualización, no se cambia la página anterior, sino se crea una página sombra con el nuevo valor y se escribe en la base de datos estable. Se actualizan los caminos de acceso de manera que los accesos posteriores se hacen a la nueva página sombra. La página anterior se retiene para propósitos de recuperación, para realizar una operación UNDO.
2. **Archivos diferenciales**. Para cada archivo F se mantiene una parte de solo lectura (FR), un archivo diferencial que consiste de la parte de inserciones DF+ y la parte de supresiones DF-. Así, el archivo completo consistirá de la unión de la parte de lectura más la parte de inserciones y a todo esto se le eliminan las supresiones realizadas.

   F = (FR È DF+) – DF-

   Todas las actualizaciones se tratan como la supresión de un valor anterior y la inserción de un nuevo valor. Periódicamente, el archivo diferencial tiene que ser mezclado con el archivo base de solo lectura.

### 7.3.3 EJECUCION DE LOS COMANDOS DEL LRM

Existen cinco comandos que forman la interfaz al LRM. Ellos son: **begin_transaction**, **read**, **write**, **commit** y **abort**. En esta sección se discutirá cada uno de ellos y se presentará el comando **recover** cuya necesidad debe ser aparente después de la discusión anterior. La ejecución de los comandos **abort**, **commit** y **recover** es completamente dependiente de los algoritmos de recuperación que se usen. Por otra parte, los comandos **begin_transaction**, **read** y **write** son independientes del LRM.

La decisión fundamental en el diseño de administrador de recuperación local, el administrador del buffer y la interacción entre estas componentes es si el administrador de buffers obedece instrucciones del LRM tales como cuando escribir las páginas del buffer de la base de datos al almacenamiento estable. Específicamente, existen dos decisiones fundamentales:

1. ¿puede el administrador de buffers escribir las páginas del buffer actualizadas por una transacción en el almacenamiento estable durante la ejecución de la misma, o tiene que esperar las instrucciones del LRM para escribirlas en la base de datos estable? A este tipo de estrategia se le conoce como *fix*/*no-fix*.
2. ¿puede el LRM forzar al administrador del buffer para que escriba ciertas páginas del buffer en la base de datos estable al final de la ejecución de una transacción? A este tipo de estrategia se le conoce como *flush*/*no-flush*.

De acuerdo a lo anterior, existe cuatro posibles alternativas: no-fix/no-flush, no-fix/flush, fix/no-flush y fix/flush.

- *Begin_transaction*. Hace que el LRM escriba un comando begin_transaction en el registro de la base de datos.
- *Read*. El LRM trata de leer los datos especificados de las páginas del buffer que pertenecen a la transacción. Si el dato no se encuentra en esas páginas, envía un comando **fetch** al administrador del buffer para que los datos sean disponibles.

- *Write*. Si el dato está disponible en los buffers de la transacción, el valor se modifica en los buffers de la base de datos. Si no se encuentra en los buffers, se envía un comando **fetch** al administrador del buffer y, entonces, se hace la actualización en la base de datos volátil.

Discutiremos ahora la ejecución de las instrucciones restantes de acuerdo a la estrategia que se siga.

**No-fix/No-flush**

- *Abort*. El administrador del buffer pudo haber escrito algunas páginas actualizadas en la base de datos estable. Por lo tanto, el LRM ejecuta un UNDO de la transacción.
- *Commit*. El LRM escribe un "end_of_transaction" en el registro de la base de datos.
- *Recover*. Para aquellas transacciones que tienen tanto un "begin_transaction" como un "end_of_transaction" en el registro de la base de datos, se inicia una operación parcial REDO por el LRM. Mientras que para aquellas transacciones que tienen solo un "begin_transaction" en el registro, se ejecuta un UNDO global por el LRM.

**No-fix/Flush**

- *Abort*. El administrador del buffer pudo haber escrito algunas páginas actualizadas en la base de datos estable. Por lo tanto, el LRM ejecuta un UNDO de la transacción.
- *Commit*. El LRM envía un comando **flush** al administrador del buffer para todas las páginas actualizadas. Se escribe un comando "end_of_transaction" en el registro de la base de datos.
- *Recover*. Para aquellas transacciones que tienen tanto un "begin_transaction" como un "end_of_transaction" en el registro de la base de datos, no se requiere una operación REDO. Mientras que para aquellas transacciones que tienen solo un "begin_transaction" en el registro, se ejecuta un UNDO global por el LRM.

**Fix/No-flush**.

- *Abort*. Ninguna de las páginas actualizadas ha sido escrita en la base de datos estable. Por lo tanto, solo se liberan las páginas correspondientes a la transacción enviándoles un comando **fix**.
- *Commit*. Se escribe un comando "end_of_transaction" en el registro de la base de datos. El LRM envía un comando **unfix** al administrador del buffer para todas las páginas a las que se les envío un comando **fix** previamente.
- *Recover*. Para aquellas transacciones que tienen tanto un "begin_transaction" como un "end_of_transaction" en el registro de la base de datos, se inicia una operación parcial REDO por el LRM. Mientras que para aquellas transacciones que tienen solo un "begin_transaction" en el registro, no se necesita un UNDO global.

**Fix/Flush**

- *Abort*. Ninguna de las páginas actualizadas ha sido escrita en la base de datos estable. Por lo tanto, solo se liberan las páginas correspondientes a la transacción enviándoles un comando **fix**.
- *Commit*. De manera atómica se tienen que hacer las siguientes acciones. Se envía un comando **flush** al administrador del buffer para todas las páginas a las que se les aplicó un comando **fix**. El LRM envía un comando **unfix** al administrador del buffer para todas las páginas a las que se les aplicó un comando **fix**. El LRM escribe un "end_of_transaction" en el registro de la base de datos.
- *Recover*. No requiere hacer trabajo alguno.

**Verificación**

La operación de recuperación requiere recorrer todo el registro de la base de datos. Así, el buscar todas las transacciones a las cuales es necesario aplicarles un UNDO o REDO puede tomar una cantidad de trabajo considerable. Para reducir este trabajo se pueden poner puntos de verificación (checkpoints) en el registro de la base de datos para indicar que en esos puntos la base de datos está actualizada y consistente. En este caso, un REDO tiene que iniciar desde un punto de verificación y un UNDO tiene que regresar al punto de verificación más inmediato anterior. La colocación de puntos de verificación se realiza con las siguientes acciones:

1. Se escribe un "begin_checkpoint" en el registro de la base de datos.
2. Se recolectan todos los datos verificados en la base de datos estable.
3. Se escribe un "fin_de_checkpoint" en el registro de la base de datos.

### 7.4 Protocolos distribuidos

Como con los protocolos de recuperación local, las versiones distribuidas ayudan a mantener la atomicidad y durabilidad de las transacciones. La ejecución de los comandos **begin_transaction**, **read** y **write** no provoca ningún cambio significativo. Sin embargo, la ejecución de las operaciones **commit**, **abort** y **recover** requieren del desarrollo de un protocolo especial para cada una de ellas en el cual participan todos los nodos de la red que intervienen en una transacción. De manera breve, a continuación se presentan algunos problemas que aparecen en sistemas distribuidos y que no se presentan en sistemas centralizados.

¿Cómo ejecutar un comando **commit** para transacciones distribuidas? ¿Cómo asegurar la atomicidad y durabilidad de las transacciones distribuidas?

Si ocurre una falla, ¿cómo pueden, los nodos que permanecen funcionando, seguir operando normalmente? Idealmente, la ocurrencia de la falla en un nodo no debe forzar a los otros nodos a tener que esperar a que se repare la falla en el nodo para terminar una transacción. A esta propiedad se le conoce como *no-bloqueante*.

De manera similar, cuando ocurre una falla ¿cómo terminar una transacción que se estaba ejecutando al momento de la falla? Idealmente un nodo con falla debería poder terminar una transacción sin tener que consultar a los otros nodos. Esta es una propiedad de *independencia* del nodo con falla con respecto a los nodos sin falla. Una recuperación independiente supone que existe una estrategia no-bloqueante en el manejo de fallas.

Para facilitar la descripción de los protocolos de confiabilidad distribuida se supondrá que en el nodo que se origina una transacción hay un proceso, llamado el *coordinador*, que ejecuta las operaciones de la transacción. El coordinador se comunica con procesos *participantes* en los otros nodos los cuales lo ayudan en las operaciones de la transacción.

#### 7.4.1 Compromisos de dos fases

El compromiso de dos fases (two-phase commit) es un protocolo muy simple y elegante que asegura la atomicidad de las transacciones distribuidas. Extiende los efectos de una operación local de commit a transacciones distribuidas poniendo de acuerdo a todos los nodos involucrados en la ejecución de una transacción antes de que los cambios hechas por la transacción sean permanentes.

Las fases del protocolo son las siguientes:

- *Fase 1*: el coordinador hace que todos los participantes estén listos para escribir los resultados en la base de datos.
- *Fase 2*: Todos escriben los resultados en la base de datos.

La terminación de una transacción se hace mediante la *regla del compromiso global*:

1. El coordinador aborta una transacción si y solamente si al menos un participante vota por que se aborte.
2. El coordinador hace un commit de la transacción si y solo si todos los participantes votan por que se haga el commit.

La operación del compromiso de dos fases entre un coordinador y un participante en ausencia de fallas se presenta en la Figura 7.9, en donde los círculos indican los estados y las líneas entrecortadas indican mensajes entre el coordinador y los participantes. Las etiquetas en las líneas entrecortadas especifican la naturaleza del mensaje.

En la Figura 7.10 se presentan las dos fases de comunicación para realizar un commit en sistemas distribuidos. El esquema presentado en la figura se conoce como centralizado ya que la comunicación es solo entre el coordinador y los participantes; los participantes no se comunican entre ellos.

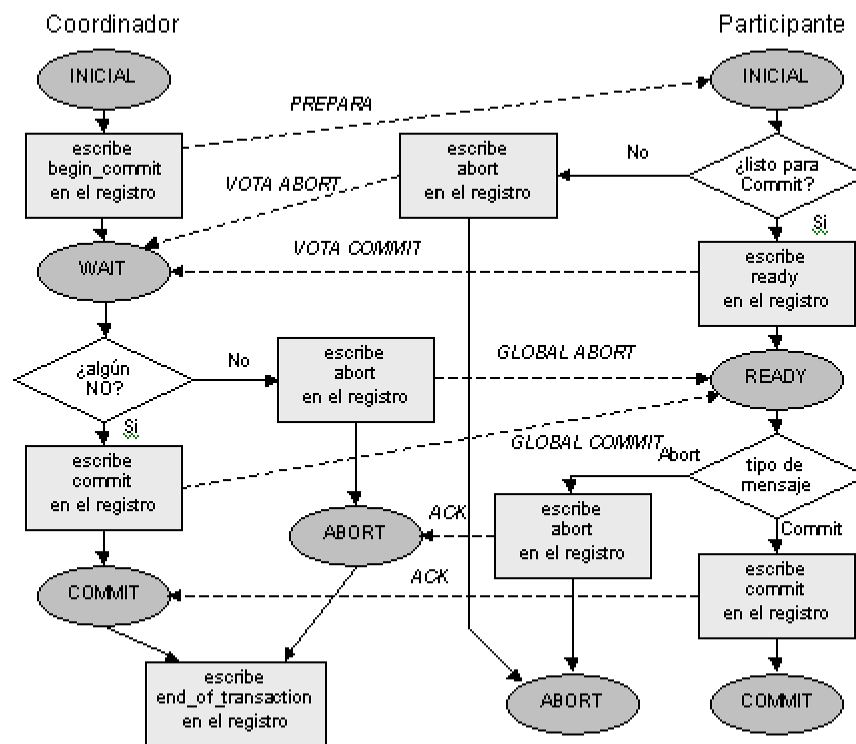



Figura 7.9. Acciones del compromiso de dos fases.

Otra alternativa es una comunicación lineal, en donde los participantes se pueden comunicar unos con otros. Existe un ordenamiento entre los nodos del sistema. La estructura se presenta en la Figura 7.11. Un participante, *P*, recibe un mensaje vote-abort o vote-commit de otro participante, *Q*. Si P no está listo para hacer un commit, envía un vote-abort al siguiente participante, *R*, y la transacción se aborta en este punto. Si P el participante está de acuerdo en hacer un commit, envía un vote-commit a *R* y entra al estado de LISTO. Este proceso continúa hasta el último participante poniendo fin a la primera fase. En la segunda fase, si el último participante está de acuerdo en hacer un commit envía un global-commit al participante anterior. En caso contrario, envía un global-abort. Así en la segunda fase, *P* recibe un mensaje de *R* informándole si se hace un global-commit o un global-abort. Este mensaje se propaga a *Q* y así hasta alcanzar al coordinador.

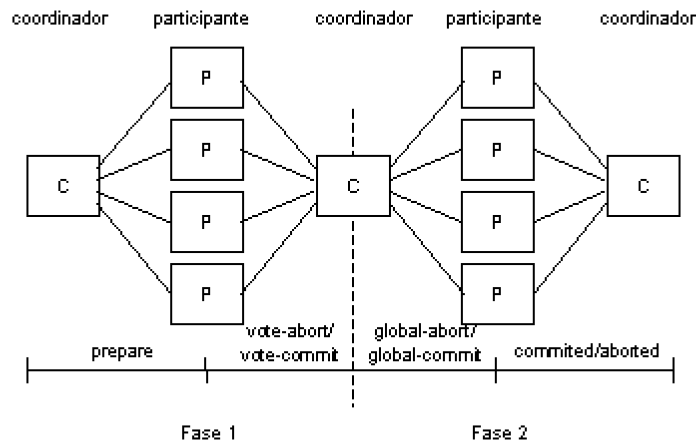



Figura 7.10. Estructura centralizada de comunicaciones para el compromiso de dos fases.

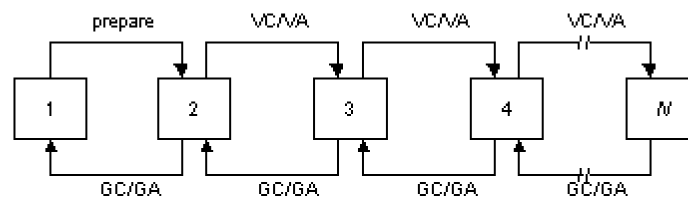



Figura 7.11. Estructura lineal de comunicaciones para el compromiso de dos fases.

Otro estructura de comunicación usual para implementar los compromisos de dos fases involucra la comunicación entre todos los participantes durante la primera fase del protocolo de manera que todos ellos alcanzan su punto de terminación en forma independiente. Esta versión, llamada la *estructura distribuida*, no requiere la segunda fase. En la Figura 7.12 se presenta la estructura distribuida en la cual el coordinador envía el mensaje de preparación a todos los participantes. Cada participante, entonces, envía su decisión a todos los otros participantes y al coordinador indicándola como un vote-commit o vote-abort. Cada participante espera los mensajes de todos los otros participantes y toma su decisión de acuerdo a la regla de compromiso global.

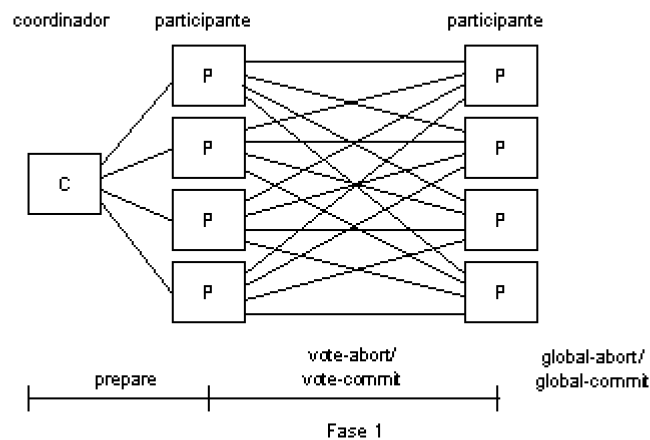



Figura 7.12. Estructura distribuida de comunicaciones para el compromiso de dos fases.

Independientemente de la forma en que se implemente el compromiso de dos fases, éste puede ser modelado por medio de un diagrama de transición de estados. En la Figura 7.13 se presentan los diagramas de transición para el coordinador y los participantes.

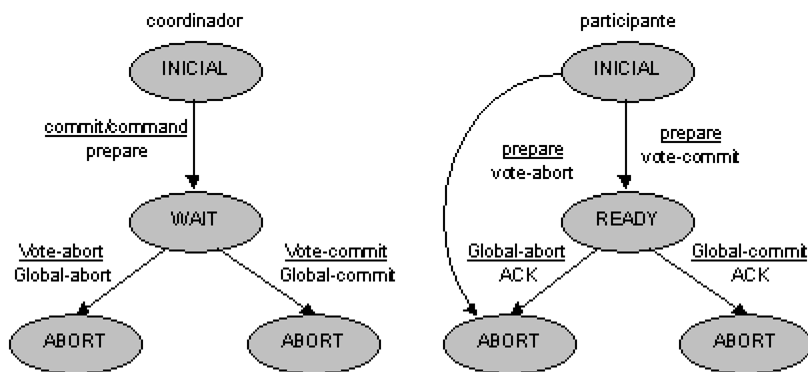



Figura 7.13. Diagrama de transición de estados para el compromiso de dos fases del coordinador y de un participante.

### 7.4.2 Fallas de nodos

En esta sección se considera cuando un nodo falla en la red. El objetivo es desarrollar un protocolo de terminación no-bloqueante que permita desarrollar protocolos de recuperación independiente. En una red puede haber muchas fallas de comunicación debido a colisiones, comunicación intermitente por interferencia, sobrecarga de trabajo, etc. La única forma de suponer que existe una falla de nodo es cuando después de un cierto periodo de tiempo (timeout) no se obtiene respuesta del mismo. Así, la discusión de los protocolos de terminación y recuperación considera el caso en que las fallas de nodo se manifiestan por ausencia de comunicación con éste durante intervalos de tiempo.

**Protocolos de terminación**

Las fallas de nodo se pueden manifestar tanto en el coordinador como en los participantes.

- **Timeouts del coordinador**

1. *Timeout en el estado WAIT*. El coordinador está esperando por las decisiones locales de los participantes. El coordinador no puede de manera unilateral realizar un commit. Sin embargo, él puede decidir abortar globalmente la transacción. En este caso, escribe un comando abort en el registro de la base de datos y envía un mensaje global-abort a todos los participantes.
2. *Timeout en los estados COMMIT o ABORT*. En este caso el coordinador no esta seguro que los procedimientos de commit o abort se han terminado por los administradores de recuperación local en todos los nodos participantes. Así, el coordinador envía mensajes global-commit o global-abort de manera repetida a todos los nodos que no han respondido y espera por su reconocimiento.

- **Timeouts de los participantes**

1. *Timeout en el estado INITIAL*. En este estado el participante está esperando por un mensaje prepare. El coordinador pudo haber fallado en el estado INITIAL por lo que el participante puede abortar de manera unilateral la transacción. Si el mensaje prepare llega en un tiempo posterior al participante, esto se puede manejar de dos formas. Puede responder con un vote-abort o simplemente ignorar el mensaje y dejar que ocurra el timeout en el lado del coordinador.

2. *Timeout en el estado READY*. En este estado el participante ha votado por realizar un commit pero desconoce la decisión global del coordinador. No se puede de manera unilateral ni hacer un commit ni hacer un abort. Así permanecerá bloqueado hasta que pueda conocer de alguien, el coordinador u otro participante, cual fue la decisión final sobre la transacción.

**Protocolos de recuperación**

En la parte anterior se discutió como implementar los compromisos de dos fases cuando se presentan fallas en nodos pero desde la perspectiva de los nodos que se mantienen trabajando. En esta parte, se examinará el otro punto de vista, esto es, como un coordinador o participante se pueden recuperar cuando sus nodos fallas y tienen que reiniciar su trabajo.

- **Fallas del coordinador**

1. *El coordinador falla en el estado INITIAL*. Esto es antes de que el coordinador ha iniciado el procedimiento para hacer un commit. Por lo tanto, cuando el coordinador reinicia tiene que empezar el proceso para hacer el commit.
2. *El coordinador falla en el estado de WAIT*. En este caso el coordinador ha enviado el comando prepare y después falla. Para recuperarse de una falla, tendrá que reiniciar el proceso de commit para la transacción desde el envío nuevamente del mensaje prepare.
3. *El coordinador falla en los estados COMMIT o ABORT*. En este caso el coordinador ha tomado una decisión y la ha informado a los participantes. Así, si todos los reconocimientos se han recibido cuando se trata de recuperar, entonces no se hace nada. De otra manera, se invoca al protocolo de terminación.

**Fallas de los participantes**

1. *Un participante falla en el estado INICIAL*. En este caso se aborta la transacción de manera unilateral cuando se trata de recuperar. Lo anterior es aceptable ya que el coordinador de la transacción debe estar en el estado INITIAL o WAIT. Si está en el primero, enviará un mensaje prepare y pasará al estado de WAIT en donde no recibirá respuesta del participante que ha fallado y eventualmente abortará globalmente la transacción.
2. *Un participante falla en el estado READY*. En este caso el participante ha informado su decisión al coordinador. Cuando se recupere, el participante en el nodo fallido puede tratar esto como un timeout en el estado de READY y manejar la transacción incompleta sobre su protocolo de terminación.
3. *Un participante falla en el estado ABORT o COMMIT*. Esos estados representan las condiciones de terminación, de manera que, cuando se recupere, el participante no necesita tomar acciones especiales.

**Casos adicionales**

Consideremos ahora los casos cuando se relajan las condiciones de atomicidad del registro y las acciones del envío de mensajes. En particular, se supone que una falla de nodo puede ocurrir después de que el coordinador o un participante ha escrito información en el registro de la base de datos pero puede enviar un mensaje. Para esta discusión es útil consultar la Figura 7.8.

1. *El coordinador falla después que un begin_commit ha sido escrito en el registro pero antes de que se envíe el mensaje prepare*. El coordinador puede considerar esto como una falla en el estado WAIT y enviar el comando prepare cuando se recupera.
2. *Un particiapante falla después de escribir un comando ready en el registro pero antes de enviar un mensaje vote-commit*. El participante ve a este caso como una falla en el estado de WAIT y procede conforme lo discutido antes.

3. *Un particiapante falla después de escribir un comando ready en el registro pero antes de enviar un mensaje vote-abort*. El participante no tiene que hacer nada cuando se recupere. El coordinador está en el estado de WAIT y no obtendrá respuesta del participante fallido lo que llevará a invocar el protocolo de terminación en donde se abortará de forma global la transacción.
4. *El coordinador falla después de registrar su decisión final (abort o commit) en el registro de la base de datos pero antes de enviar su mensaje global-abort o global-commit a los participantes*. El coordinador trata a éste como el caso 3, mientras que los participantes lo tratan con un timeout en el estado de READY.
5. *Un participante falla después de registrar un abort o commit pero antes de enviar un mensaje de reconocimiento al coordinador*. El participante trata a éste como en el caso 3, mientras que el coordinar lo maneja como un timeout en el estado COMMIT o ABORT.