MANEJO DE TRANSACCIONES CENTRALIZADAS Y DISTRIBUIDAS

COLECTIVO DE AUTORES

Edición: Liset Ravelo Romero
Corrección: Fernando Gutiérrez Ortega
Diagramación: Roberto Suárez Yera

ISBN: 978-959-250-364-9

EDITORIAL
Feijóo

Editorial Samuel Feijóo, Universidad Central “Marta Abreu” de Las Villas, Carretera a Camajuaní, km 5 ½, Santa Clara, Villa Clara, Cuba. CP 54830

RESUMEN

Este material cubre elementos teóricos relacionados con los Sistemas Manejadores de Bases de Datos. Se parte de la arquitectura de tales sistemas, se identifican sus componentes, y funciones, asi como las interrelaciones e interacciones entre ellos. Se hace una caracterización de las transacciones y su gestión en ambientes centralizados y distribuidos, además de un estudio detallado de los conceptos (algoritmos y protocolos) sobre el control de la concurrencia en sistemas centralizados y distribuidos. Por último se tratan los elementos relacionados con la optimización de consultas. Esta monografía puede servir como apoyo a la docencia de pregrado y postgrado en temas avanzados de bases de datos.

ÍNDICE

INTRODUCCIÓN

Las actualizaciones en las bases de datos se originan generalmente por eventos del mundo real, y usualmente involucran varias acciones organizadas en unidades denominadas transacciones. Los Sistemas Manejadores de Bases de Datos son responsables de que se ejecuten todas las sentencias en una transacción se ejecuten, o ninguna, en caso de que la aplicación aborte o se produzca un fallo de hardware durante la ejecución de la misma. El manejo de transacciones se ocupa de mantener las bases de datos en estados consistentes aun cuando ocurran accesos concurrentes o fallas.

Actualmente, los Sistemas Manejadores de Bases de Datos Centralizados tienen un gran impacto, aunque se espera que en los próximos años sean una curiosidad antigua, y la mayoría de las organizaciones migren hacia Manejadores Distribuidos. La distribución es muy importante porque se corresponde mejor con la estructura de las organizaciones ampliamente distribuidas de hoy día, permitiendo compartir datos en diferentes sitios con más capacidad para enfrentar y resolver los problemas que se presentan.

Este material cubre elementos teóricos relacionados con los Sistemas Manejadores de Bases de Datos. Se parte de la arquitectura de tales sistemas, se identifican sus componentesy funciones, asi como las interrelaciones e interacciones entre ellos. Se hace una caracterización de las transacciones y su gestión en ambientes centralizados y distribuidos, además de un estudio detallado de los conceptos (algoritmos y protocolos) sobre el control de la concurrencia en sistemas centralizados y distribuidos. Por último se tratan los elementos relacionados con la optimización de consultas. Esta monografía puede servir como apoyo a la docencia de pregrado y postgrado en temas avanzados de bases de datos.

CAPÍTULO 1. INTRODUCCIÓN A LOS SISTEMAS MANEJADORES DE BASES DE DATOS

Los sistemas de cómputo están sufriendo una revolución. Desde 1945 ─cuando empezó la era de la computadora moderna─ hasta cerca de 1985, las computadoras eran grandes y caras por lo que la mayor parte de las organizaciones tenían una pequeña lo un puñado de cantidad de estas y, por carecer de una forma para conectarlas entre si, operaban por lo general de forma independiente entre sí. Sin embargo, a partir de la mediado de los años 80 dos avances tecnológicos comenzaron a cambiar esta situación.

1- Desarrollo de poderosos microprocesadores con poder de cómputo de una gran mainframe por una fracción de su precio; mejoras ocurridas desde computadoras que costaban 10 millones de USD y ejecutaban una instrucción/segundo hasta máquinas que cuestan 1 000 USD y ejecutan 10 millones de instrucciones/segundo. Como puede verse, la ganancia precio/rendimiento es del orden de 10^{11}.

2- Introducción de redes de área local (LAN) de alta velocidad, gran número de CPUs conectadas mediante una red de alta velocidad nombrada sistemas distribuidos, en contraste con los sistemas centralizados anteriores o sistemas con un sólo procesador.

Sin envargo hay un problema: el software, Los sistemas distribuidos necesitan un software radicalmente distinto al de los sistemas centralizados. En particular, los sistemas operativos necesarios para estos sistemas distribuidos están apenas en una etapa de surgimiento. Se han dado algunos primeros pasos, pero todavía existe un largo camino por recorrer.

Existen varias definiciones sobre qué es un *sistema distribuido* pero ninguna es satisfactoria ni está de acuerdo con las demás. Para los propósitos de este material

es suficiente dar a continuación se ofrece una breve caracterización, suficiente de acuerdo a los propósitos de este texto.

Un sistema distribuido es una colección de computadoras independientes que aparece ante los usuarios del sistema como una única computadora cualquier sistema en el que varias CPUs conectadas entre sí trabajan de manera conjunta.

Esta definición tiene dos aspectos esenciales:
1- Hardware: máquinas autónomas
2- Software: los usuarios piensan que el sistema es como una única computadora.

Ejemplos de Sistemas Distribuidos:

a) Red en un departamento de una universidad o compañía, donde además de las estaciones de trabajo personal pueden existir algunos procesadores en el cuarto de máquinas no asignados a usuarios específicos sino que se usan de manera dinámica, según se necesiten. Pueden tener un sistema de archivo único, con todos los archivos accesibles desde todas las máquinas de la misma forma y con el mismo nombre de ruta de acceso. Además, cuando un usuario escribe un comando, el sistema podrá buscar el mejor lugar para ejecutarlo, tal vez en la propia estación, o en una inactiva que pertenezca a otra persona, o en uno de los procesadores no asignados en el cuarto de máquinas.

b) Un banco con cientos de sucursales por todo el mundo, cada oficina tiene una computadora maestra para guardar las cuentas locales y el manejo de las transacciones locales. Además, cada computadora puede comunicarse con las de otras sucursales y con una gran computadora central en las oficinas centrales. Si se pueden realizar las transacciones sin importar dónde se encuentren el cliente o la cuenta, si los usuarios no observan diferencias entre este sistema y el antiguo sistema centralizado que ha reemplazado, también se puede considerar como un sistema distribuido.

El hecho de poder construir sistemas distribuidos no significa necesariamente que sea buena idea. Después de todo, con la tecnología actual, es posible colocar 4 unidades de disco flexible en una computadora personal, pero lo importante ho es que no tendría sentido. La economía es la fuerza motriz real de la tendencia hacia la descentralización. A continuación, se resumen algunas ventajas de los sistemas distribuidos sobre los sistemas centralizados.

ELEMENTO	DEFINICIÓN
Economía	Los microprocesadores ofrecen mejor proporción que los mainframes en cuanto a precio/rendimiento. Una colección de microprocesadores no sólo proporciona mejor precio/rendimiento que un mainframe sino que pueden tener mejor desempeño del que podría ofrecer cualquier mainframe a cualquier precio.
Velocidad	Un sistema distribuido puede tener mayor poder de cómputo que en mainframe.
Distribución Inherente	Algunas aplicaciones utilizan máquinas que están separadas a cierta distancia. Por ejemplo, en una red de supermercado que recibe suministros de granjas locales, la toma de decisiones es local en lugar de central. Se puede hacer ver el sistema como una computadora para los programas de aplicación, pero implantado de manera descentralizada con una computadora por tienda. Otro ejemplo puede ser el trabajo cooperativo apoyado por computadoras, juegos, etc.
Confiabilidad	Si una máquina se descompone, el sistema puede sobrevivir como un todo. Para el caso de aplicaciones críticas como control de reactores nucleares o aviación, el uso de un sistema distribuido para lograr mayor confiabilidad puede ser un factor dominante.
Crecimiento por incremento	Se puede añadir poder de cómputo en pequeños incrementos. Si una compañía compra un mainframe y aumenta la carga de trabajo al prosperar, deja de ser adecuado un cierto momento y hay que

	sustituirlo o añadir otro si no existe uno mayor, lo cual puede representar un duro golpe a las operaciones de la compañía.

Seguidamente se listan algunas ventajas de los sistemas distribuidos sobre las computadoras aisladas.

ELEMENTO	DEFINICIÓN
Datos Compartidos	Permite que varios usuarios tengan acceso a una base de datos común ya que muchos usuarios necesitan compartir ciertos datos, por ejemplo, los empleados de línea aérea (reservaciones) necesitan tener acceso a la base de datos maestra de vuelos y reservaciones existentes. Es necesario que las máquinas estén conectadas entre sí. La conexión conduce a un sistema distribuido.
Dispositivos Compartidos	Varios usuarios comparten periféricos caros. Por ejemplo, impresoras láser a color, equipos de fotocomposición, dispositivos de almacenamiento masivo como cajas ópticas.
Comunicación	Facilita la comunicación persona a persona, por ejemplo, el e-mail es más rápido que el correo postal, no requiere que ambas partes estén disponibles al mismo tiempo como el teléfono y a diferencia del FAX los documentos se pueden editar, reordenar, almacenar y manejar mediante procesadores de texto.
Flexibilidad	Difunde la carga de trabajo entre las máquinas disponibles en la forma más eficaz en cuanto a costo. La perdida de unas cuantas máquinas se puede compensar si se permite a las personas que ejecuten sus trabajos en otra parte.

Aunque se han expresado un conjunto de ventajas, es necesario decir que también se tienen algunas desventajas en los sistemas distribuidos como se puede observar a continuación

ELEMENTO	DEFINICIÓN
Software	Existe poco software para sistemas distribuidos en la actualidad, no se tienen muchas experiencias en el diseño, implementación y uso del

	software distribuido. Es necesario responder algunas preguntas: ¿Qué tipos de sistemas operativos, lenguajes de programación y aplicaciones son adecuadas para estos sistemas? ¿Cuánto deben saber los usuarios de la distribución? ¿Qué tanto debe hacer el sistema y qué tanto deben saber los usuarios?
Redes	La red se puede saturar o causar otros problemas. Si hay saturación hay que remplazar o añadir otra, tener cables costosos o remplazar tarjetas de red, por ejemplo, por fibra óptica. La dependencia de la red puede negar algunas ventajas.
Seguridad	Un acceso sencillo también se aplica a datos secretos. El hecho de que los datos son fáciles de acceder es una ventaja, pero se puede convertir en un arma de dos filos. Si se puede acceder a todos los datos del sistema, entonces alguien puede acceder a sean con los que no tiene nada que ver. Para mantener el secreto es preferible una computadora personal aislada, sin conexiones de red, y mantenida en un cuarto cerrado y seguro.

A pesar de los problemas potenciales, muchas personas sienten que las ventajas tienen mayor peso que las desventajas y se espera que los sistemas distribuidos tengan cada vez mayor importancia en los años venideros. Es probable que en unos cuantos años, gran parte de las organizaciones conecten la mayoría de sus computadoras a extensos sistemas distribuidos con el fin de proporcionar un mejor servicio, más barato y más conveniente a los usuarios. Es probable que una computadora aislada en una empresa de tamaño medio o grande o alguna otra organización ya no exista dentro de 10 años. Aunque el hardware es importante, el software lo es más.

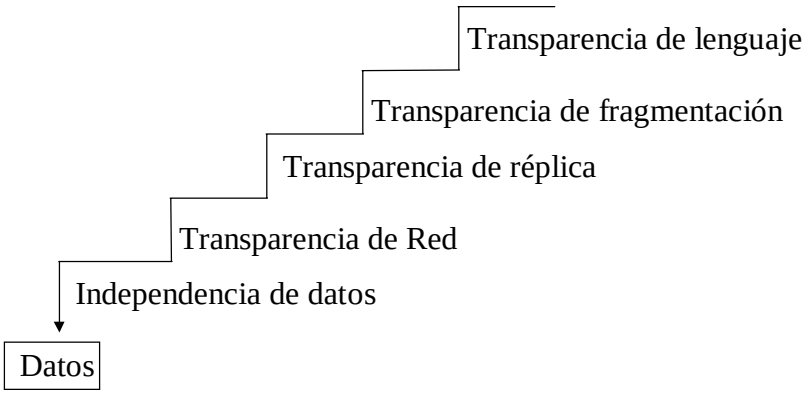
ARQUITECTURA DE LOS SISTEMAS MANEJADORES DE BASES DE DATOS

La arquitectura de los Sistemas Manejadores de Bases de Datos (SMBD) se encarga de definir un punto de referencia, que es una visión idealizada, para tales sistemas, y consiste en identificar componentes, especificar la función de cada componente, y definir los interrelaciones e interacciones entre ellos. Los aspectos relacionales en los niveles de transparencia constituyen metas a lograr. Un sistema transparente oculta detalles de implementación a los usuarios. Las capas de transparencia son:

1. Independencia de datos: es fundamental y tiene que ver con la inmunidad de las aplicaciones de usuarios de cambiar la definición y organización de los datos y viceversa. Hay dos tipos básicos: independencia lógica e independencia física.
1.1. Independencia lógica de datos: inmunidad de los programas ante cambios en la estructura lógica de datos.
1.2. Independencia física de datos: ocultar los detalles de las estructuras de almacenamiento a las aplicaciones. Si se cambia la organización (acceso secuencial, indizado, aleatorio, etc.) no debe ser necesario cambiar los programas de aplicación.
Estado actual: En los SMBDs de microcomputadoras no aparecen altos grados de independencia, es preciso descargar, cambiar, y luego cargar. Unos productos comerciales proveen mejor independencia que otros.
2. Transparencia de Red: el usuario no debe manejar detalles operacionales de la red. Para las Bases de Datos Distribuidas (BDD) la transparencia sobre la distribución está relacionada con las transparencias de localización y de nombrado
3. Transparencia de réplica: la cuestión está en decidir si replicar o no, y cuántas copias tener. Puede causar problemas en la actualización ya que si las aplicaciones son fundamentalmente orientadas a la actualización, no es buena idea tener copias. Para el usuario debe parecer como si hubiera una sola copia, aunque en algún momento define cuántas copias va a tener.
4. Transparencia de fragmentación: la fragmentación reduce efectos negativos de las réplicas, cada fragmento se maneja como otra relación. Se usa con frecuencia por razones de desempeño, disponibilidad e integridad/fiabilidad. Cuando se fragmenta

hay que tener en cuenta que hay solicitudes que manejan relaciones completas y ahora deben ser ejecutadas en subrelaciones.

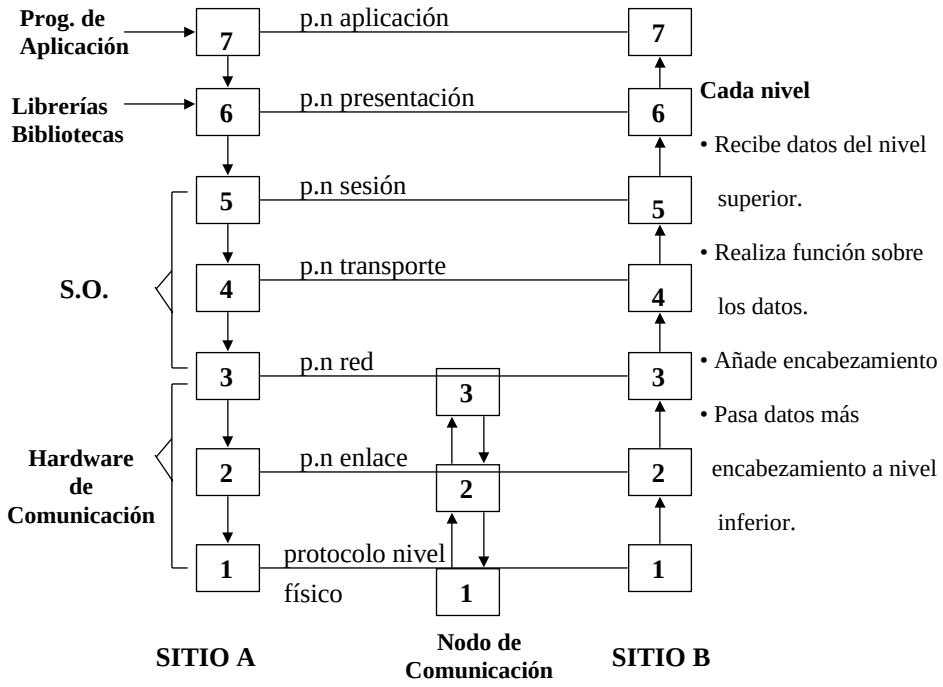
¿Quién produce estas transparencias? El SMBD que actúa como sistema operativo integrado y manejador de bases de datos.



ESTANDARIZACIÓN DE LOS SMBD

1. Basados en componentes: Se definen los componentes junto a sus interrelaciones. Ejemplo: Estándar preparado por Computer Corporation of America para el National Bureau of Standards (CCA para NBS)

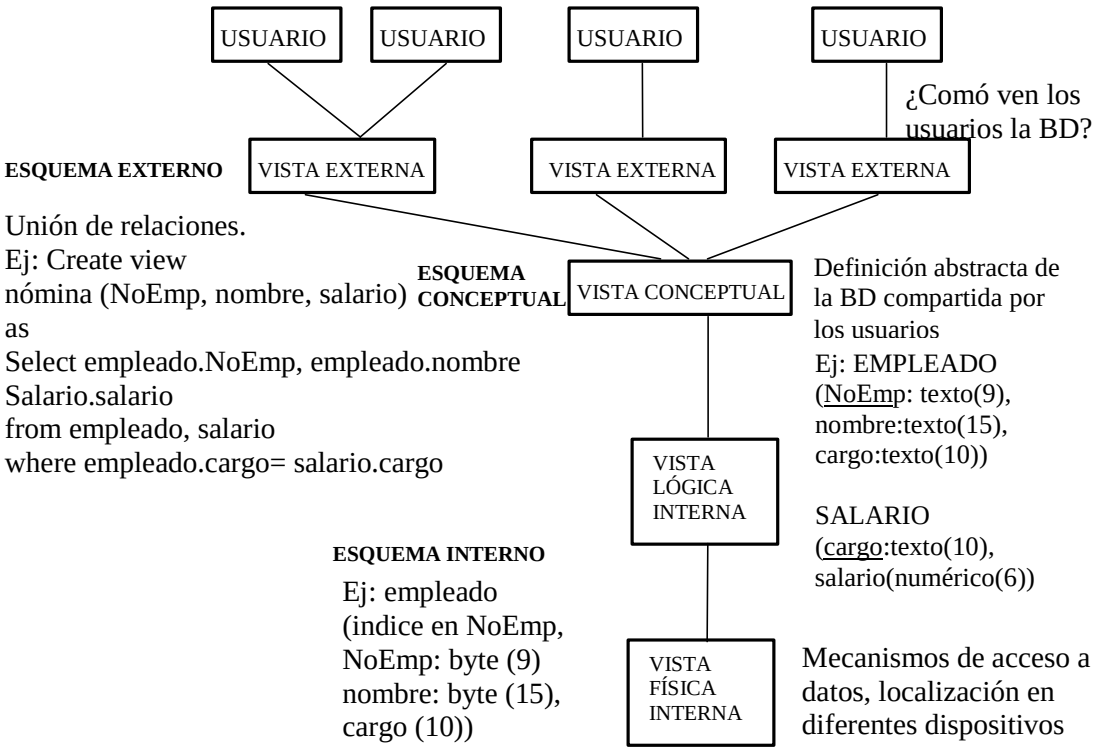
2. Basados en funciones: se identifican clases de usuarios y se definen las funciones que el sistema desarrollará para cada clase identificada. Ejemplo: Estructura jerárquica para clases de usuarios, como es el caso de ISO/OSI.



3. Basado en datos: se identifican en diferentes tipos de datos y se especifica una arquitectura que define las unidades funcionales que usan los datos.

La tercera opinión es la preferible a estandarizar, también deben ser escritos los modelos funcionales. Ejemplo: ANSI/SPARC (El grupo de estudios de SMBD crea el comité de planificación y requerimiento de estándares). Se deben usar 3 esquemas juntos para definir una arquitectura, esto aparece en la figura siguiente.

La tecnología de sistemas de bases de datos distribuidos (SBDD) es uno de los desarrollos recientes de mayor envergadura en el área SBD. Se espera que en los próximos años los manejadores de BD centralizados sean una curiosidad antigua y la mayoría de las organizaciones “migren” hacia manejadores de BDD.

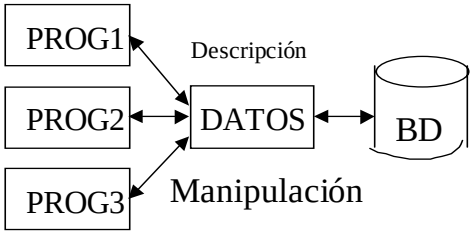


La distribución es muy importante porque se corresponde mejor con la estructura organizacional de las empresas ampliamente distribuidas de hoy en día.Los SBDD son es más confiables y respóndedor, es los datos pueden ser entrados y almacenados cuando se generan sin necesidad de movimiento físico (manual), decrece continuamente el costo de la memoria y elementos de procesamiento, permiten crecimiento incremental, reducen el impacto de cambios en las funciones de procesamiento de datos, permiten compartir datos en diferentes sitios (que previamente no eran compartibles) y tienen más capacidad de enfrentar y resolver los problemas que se presentan actualmente usando la técnica de “divide y vencerás”.

Un Sistema Manejador de Bases de Datos Distribuidas (SMBDD) es un sistema de software que permite el manejo de los sistemas de bases de datos distribuidas y hace que la distribución sea transparente para los usuarios.

Tecnología de SBDD = Tecnología + Redes de computadoras

Datos definidos y administrados centralmente



Independencia de datos (prog. de aplic. inmune a cambios en organizaciones física o lógica y viceversa

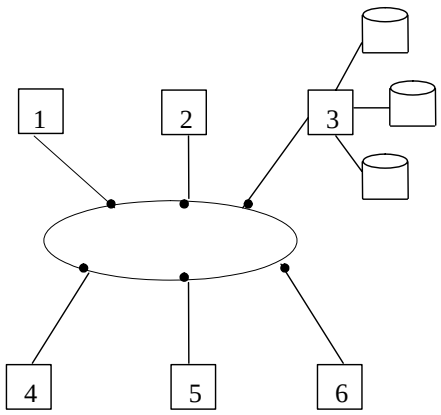
Promueve un modo de trabajo que va en contra de todos los esfuerzos de centralización

Integrar datos operacionales de empresas y proporcionar acceso controlado y centralizado a los datos

Objetivos de la tecnología BDD

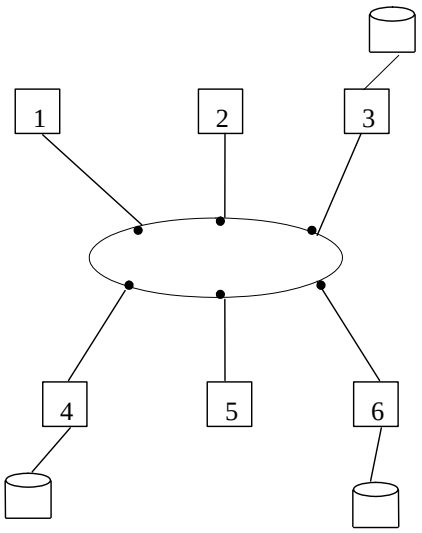
INTEGRAR, NO CENTRALIZAR

ENFOQUE CENTRALIZADO



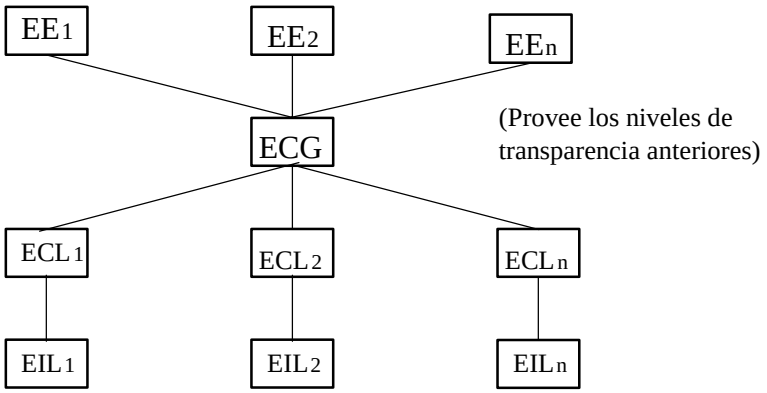
Sitio 3 : Maneja centralmente la BD.
(solicitudes enrutadas hacia acá)

ENFOQUE DISTRIBUIDO



En este documento, para el caso de los SMBDD, sólo se tratan extensiones de ANSI/SPARC, cuyos puntos de partida son:

Organización física diferente de los datos en las máquinas. EIL - esquema interno individual (local) en cada sitio.
Esquema conceptual global. ECG - describe estructura lógica de los datos de todos los sitios.
Esquema conceptual local. ECL - fragmentos y réplicas. ECG = $ECL_1 \cup ECL_2 \cup ... \cup ECL_n$
Esquema externo. EE - soporta accesos a la base de datos.



Arquitectura de referencia de las bases de datos distribuidas

Las transparencias se proveen de la forma siguiente:

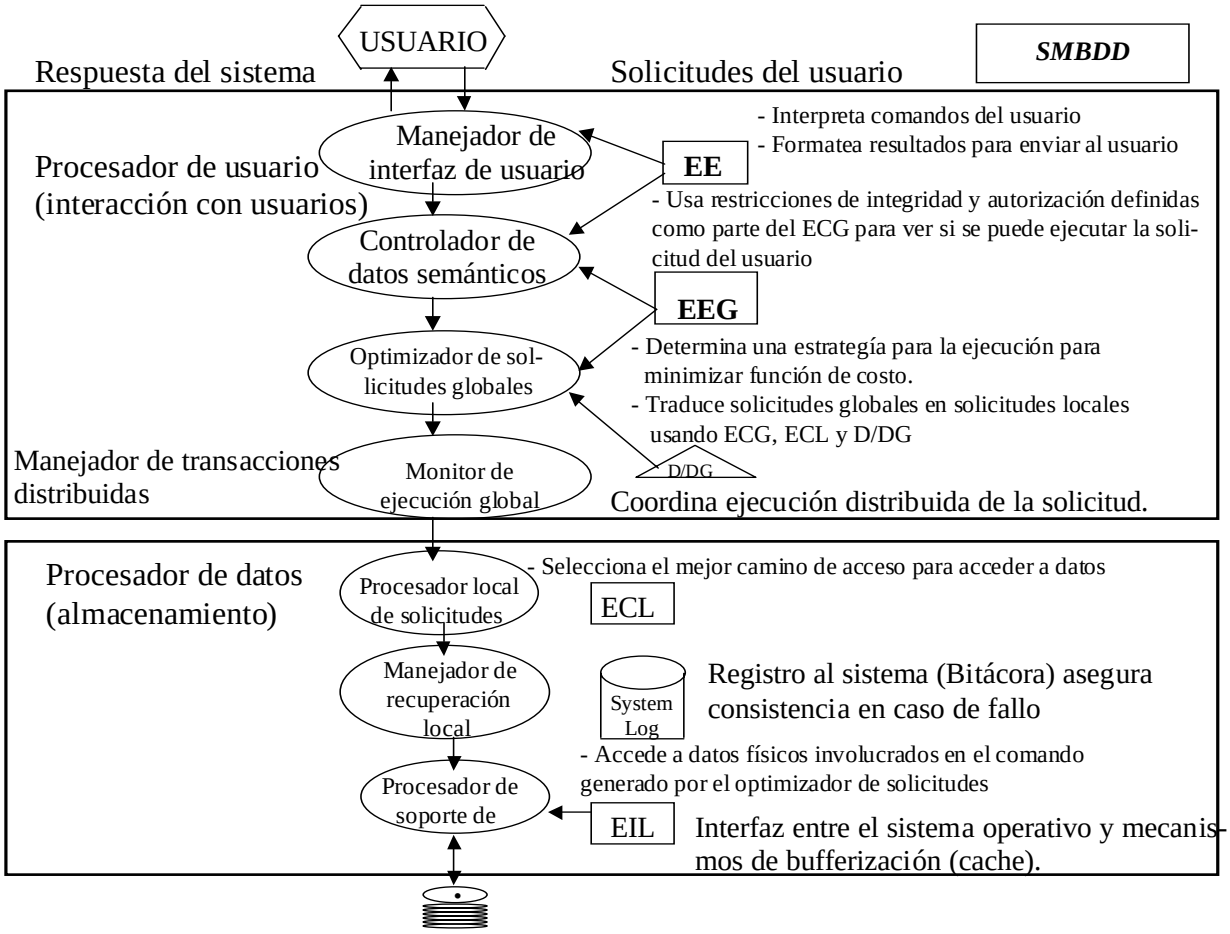
1. Independencia de datos: al ser extensión del ANSI/APARC se provee naturalmente.
2. Transparencia de réplica y localización: se provee por definición de esquemas conceptuales locales y globales y la relación entre ellos.
3. Transparencia de red: se provee por definición de ECG. El SMBD traduce solicitudes globales en conjunto de solicitudes locales que son ejecutadas por los SMBDD.

Si se añade un directorio/diccionario global (D/DG) se tienen mapeos globales, los locales se pueden desarrollar por (D/DL) directorios locales. Estas componentes se integran mediante funciones del SMBD global.

COMPONENTES DE LOS SMBDD

– Procesador de usuario: maneja interacción con los usuarios.

– Procesador de datos: maneja almacenamiento, etc.



VENTAJAS DE LOS SBDD

1. Autonomía local los usuarios tienen control local sobre los datos ubicados en un sitio de trabajo. Pueden colocar y hacer cumplir políticas locales en cuanto al uso de los datos.
2. Ejecución mejorada: los datos usados regularmente se encuentran próximos a los usuarios, y dado el paralelismo inherente de los sistemas distribuidos es posible mejorar el desempeño de accesos a las bases de datos. Cada sitio manipula una porción de la base de datos, las disputas por la CPU y servicios de E/S no son tan severos como en BD centralizadas. Los datos recuperados por una transacción pueden ser almacenados en varios sitios, siendo posible ejecutar en paralelo. Ahora, si el mantenimiento de artículos se hace centralmente en oficinas principales con acceso a otros sitios, entonces el desempeño se verá afectado.
3. Fiabilidad/ disponibilidad mejorada: si hay réplicas, el fallo de la comunicación o pérdida de un sitio no hace que sea imposible obtener los datos. Puede proveer servicio aunque un poco limitado más no inoperable.
4. Economía: en términos de comunicación, si hay fuerte interacción con datos puede ser más económico particionar la aplicación y hacer procesamiento local. La cuestión es el costo de comunicación de datos. Normalmente; cuesta menos poner juntas un conjunto de computadoras pequeñas con el poder equivalente de una sola máquina.
5. Expandibilidad: es más fácil acomodar incrementos en el tamaño de la base de datos. La expansión puede ser manipulada adicionando poder de procesamiento.
6. Compartimentación: las organizaciones con operaciones geográficamente distribuidas normalmente almacenan los datos en forma distribuida. En un sistema de información no distribuido es imposible compartir los datos y recursos.

DESVENTAJAS DE LOS SBDD

1. Falta de experiencia: los SBDD de propósito general no son usados comúnmente. Lo que se tiene son prototipos o sistemas hechos para una aplicación, por ejemplo, los sistemas de reservaciones aéreas. Las soluciones propuestas a varios problemas no han sido probadas en ambientes de operación reales.
2. Complejidad: Los problemas de los SBDD son más complejos que los de SBDC pues incluyen los de estos últimos y otro conjunto más a resolver.
3. Costo: se requiere hardware (mecanismos de comunicación), por tanto el costo de este se incrementa el costo del hardware, sin embargo hay una tendencia a disminuir. Puede secesitar software más complejo adicional. El esfuerzo de réplica (facilidades de cómputo) tiene como resultado un incremento de personal en operaciones de procesamiento de datos.
4. Distribución del Control: la distribución crea problemas de sincronización y coordinación. Si no se adoptan políticas adecuadas puede convertirse en riesgosa.
5. Seguridad: no se tiene el control sobre los datos que ofrecen las BDC. Las redes tienen sus propios requisitos de seguridad.
6. Definición de Cambio: los grandes sistemas no distribuidos que se encuentran en operación son díficiles de llevar a distribuidos al no existir herramientas ni metodologías que ayuden. Las investigaciones en bases de datos heterogéneas y una integración de bases de datos pueden vencer estas dificultades.

Existen algunos factores que complican aún más esta situación. Algunos de estos son:
– Al poder tener réplica, el SBDD debe seleccionar una de las copias en caso de recuperación y reflejar las actualizaciones en cada copia.

– Si falla algún sitio, el sistema debe asegurar que en caso de estarse realizando una actualización, sus efectos sean reflejados en los sitios de fallo al recuperarse el sistema del fallo.
– Al no tener información instantánea de las acciones de otros sitios, la sincronización de transacciones de múltiples sitios es mucho más difícil que en un sistema centralizado.

SISTEMAS DE BASES DE DATOS MÚLTIPLES

En estos años se han desarrollado nuevas aplicaciones de bases de datos que, a menudo requieren datos de una gran variedad de bases de datos ya existentes, localizadas en distintos entornos de hardware y software heterogéneos. La manipulación de datos localizados en una base de datos heterogénea requiere una capa de software adicional en la parte superior de los sistemas de base de datos existentes. A esa capa de software se le denomina sistema de bases de datos múltiples (SBDM). Los sistemas de bases de datos locales pueden emplear diferentes modelos lógicos, definición de datos y lenguajes de manipulación de datos, y pueden diferir en sus controles de concurrencia y mecanismos de gestión de transacción.

Un sistema de base de datos múltiple ofrece muchas ventajas. Extiende significativamente la capacidad de los usuarios, permitiendo es a los estos acceder y compartir datos sin añadir la carga de tener que aprender la complejidad de los diferentes sistemas manejadores de bases de datos. Los programas y procedimientos permanecen operativos en el entorno integrado de base de datos múltiple.

La estrategia de diseño que se sigue en el proceso de diseño de sistemas de bases de datos múltiples es ascendente (Bottom-up). En otras palabras, las bases de datos individuales que existen se integran mediante el diseño de un esquema conceptual global, hay otras metodologías que para la integración consideran esquemas

externos locales en lugar de esquemas conceptuales locales ya que puede no ser deseable incorporar el esquema conceptual local completo en la base de datos múltiple.

Traducción de esquemas

La traducción de esquemas consiste en el mapeo de un esquema a otro. Puede no ser necesario hacerla si se hace durante la etapa de integración. Es decir, se pueden combinar la traducción y la integración en un integrador con la información de toda la base de datos global. Este integrador debe solucionar un modelo formal para hacer la integración. Los más estudiados son el modelo relacional y el modelo de datos Entidad-Relación, siendo este último el más popular. Si las relaciones son más complejas el mapeo no es tan trivial.

Integración de esquemas

La integración de esquemas genera el esquema conceptual global. Este integra componentes de cada esquema intermedio identificando los componentes de una BD que están relacionados con otros y seleccionando la mejor forma de representación. Hay varias metodologías desarrolladas: la binaria que manipula dos esquemas a la vez, y la n-aria que manipula más de dos esquemas a la vez. La idea de integrar es alcanzar una vision unificada de los datos, donde se utiliza un modelo común de datos. Desde el punto de vista práctico, la elección conduce al modelo relacional, con SQL como lenguaje común de consulta. De hecho, en la actualidad existen varios sistemas prototipos disponibles para permitir consultar SQL a un SMBD no relacional. El esquema conceptual común puede obtenerse como se ve más abajo. El esquema de integración es una tarea muy complicada debido a la heterogeneidad semántica.

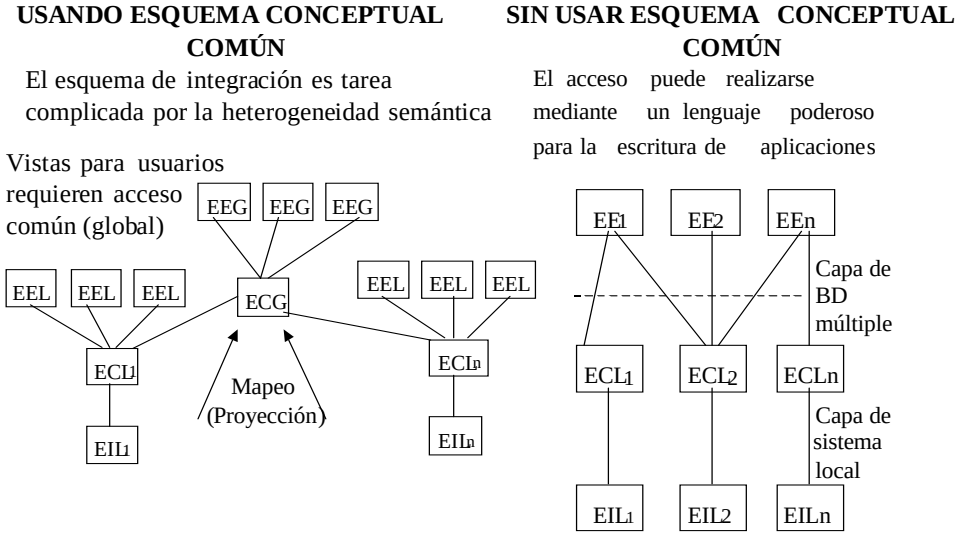
Los SMBD locales pueden emplear diferentes modelos lógicos, definiciones de datos, lenguajes de manipulación de datos, controles de concurrencia, mecanismos

de gestión de transacción. Un SMBD múltiple crea la ilusión de una integración lógica de BD sin requerir integración física. Entre sus ventajas se pueden encontrar las siguientes:

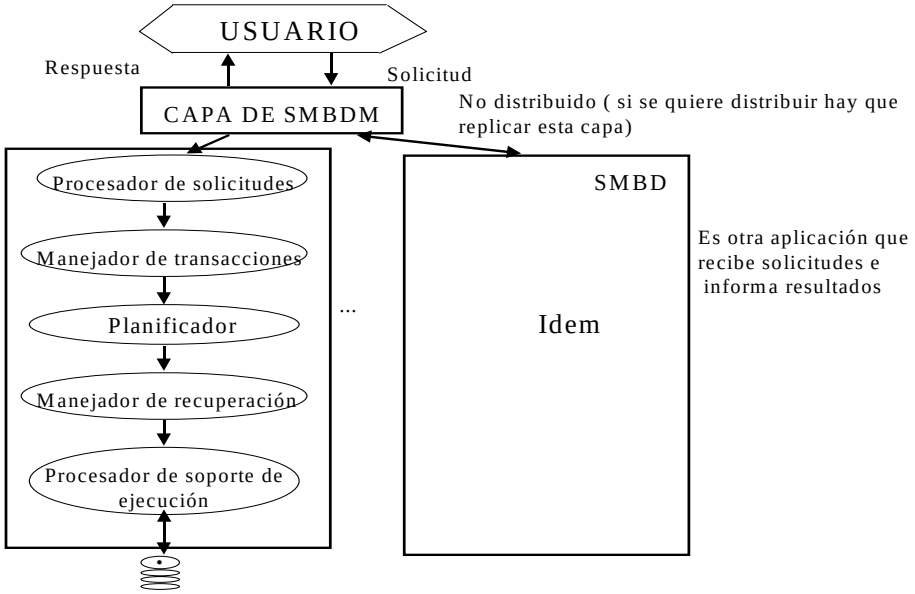
– Extiende la capacidad del usuario permitiendo acceder y compartir datos sin añadir carga de tener que aprender la complejidad de los diferentes SMBD.

– Los programas y procesamientos permanecen operativos en el entorno integrado de BD múltiple.

Arquitectura de BD múltiples



Componentes de los SBD Múltiples



ÁREAS DE PROBLEMAS

1. Diseño de la BDD: se trata de decidir cómo ubicar la BD en diferentes sitios. Para esto existen alternativas: particionadas (partes disjuntas en cada sitio); replicadas totalmente (toda la BD en cada sitio), y replicadas parcialmente (cada partición de la BD se almacena en más de un sitio pero no en todos). Se tienen los principios siguientes: fragmentación (separación de la BD en partes o fragmentos), y distribución (distribución óptima de los fragmentos).

2. Procesamiento distribuido de solicitudes: diseño de algoritmos que analizan las solicitudes y las convierten en una serie de operaciones de manipulación de datos. Este considera: distribución de los datos, costos de comunicación, y falta de información disponible localmente. Con el objetivo de optimizar, si se puede, se usa paralelismo para mejorar el desempeño en la ejecución de transacciones.

3. Manejo distribuido de directorios: almacena una descripción de elementos y localizaciones de la BD. Este puede ser: global a toda la BD o local a cada sitio, cetralizado en un sitio o distribuido en varios sitios, y de una copia o muchas copias.

4. Control de concurrencia distribuida: es el tema más estudiado. Involucra: sincronización del acceso a la BDD, y mantenimiento de la integridad de la BD (consistencia de todas las copias). Las alternativas de solución son: pesimista (sincroniza las solicitudes antes de comenzar a ejecutarlas), y optimista (ejecuta las solocitudes y chequea si la ejecución comprometió la consistencia de la BD). Los mecanismos más usados son: bloqueo (basado en la exclusión en el acceso a los datos), uso de ordenación por estampillas (*timestamps* u hora de entrada); o secuenciador (ejecución de transacciones en algún orden). Hay algoritmos híbridos que combinan estos mecanismos.

5. Manejo de abrazo mortal (*deadlock*) distribuido: ompetencia de procesos por acceder simultáneamente a un conjunto de recursos (datos). Si está basado en bloqueos conduce a deadlocks. Se pueden aplicar los mismos mecanismos para prevenir, detectar/recuperar datos conocidos en los sistemas operativos.

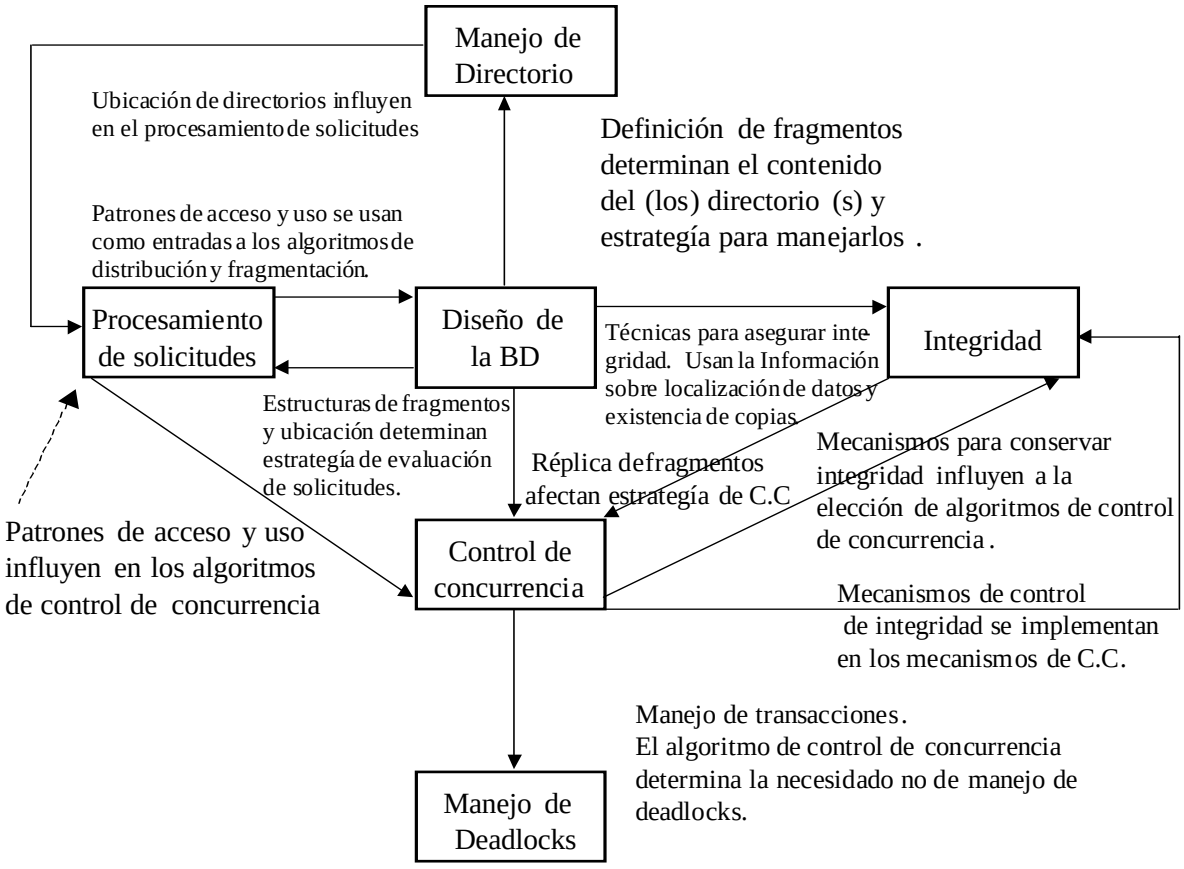
6. Integridad de los SMBDD: esto no es algo que aparece automáticamente, deben suministrarse mecanismos que aseguren la consistencia, detecten las fallas y las recuperen (actualizar la BD en los sitios de fallas).

7. Soporte de sistemas operativos: los servicios ofrecidos por los sistemas operativos no son apropiados para los SMBDDS. Hay que tratar con varias capas de software de red para realizar las actividades.

8. Bases de datos heterogéneas: se trata de los sistemas de BD múltiples. Hay especificaciones complementarias de cada uno de estos aspectos. Para los heterogéneos existen modelos de datos (estructuras lógicas), y lenguajes de manipulación (mecanismos de acceso a datos). Es necesario disponer de

mecanismos de traducción en los SBD que involucren formas canónicas (traducción de datos) y plantillas de programas (traducción de instrucciones de manipulación de datos)

9. Relación entre problemas: Puede ser interpretada del gráfico siguiente.



CAPÍTULO 2. INTRODUCCIÓN AL MANEJO DE TRANSACCIONES

La primitiva de acceso básica que se ha considerado ha sido la solicitud, sin embargo aún faltan por considerar aspectos importantes en su procesamiento tales como accesos simultáneos (concurrentes) a un mismo elemento de datos, posibilidad de fallos durante la ejecución, etc. O sea, no se maneja la noción de ejecución consistente o cómputo recuperable asociado con el concepto de una solicitud.

El concepto de transacción es usado en bases de datos como una unidad básica de consistencia y recuperación. Las solicitudes son ejecutadas como transacciones una vez que sus estrategias de ejecución se determinan y se trasladan en operaciones primitivas sobre la base de datos.

Una BD está en un estado consistente si satisface todas las restricciones de integridad definidas para la misma. Como pueden ocurrir cambios de estados debido a actualizaciones, inserciones y modificaciones, se debe asegurar que la BD no caiga en un estado inconsistente. La BD puede estar en un estado inconsistente temporal durante la ejecución de una transacción pero debe asegurarse su consistencia una vez que termine su ejecución, exitosa o no.

La consistencia de las transacciones se refiere, por otra parte, a las acciones de transacciones concurrentes; la BD debe permanecer en un estado consistente aun si hay un número de aplicaciones de usuarios que acceden concurrentemente a la BD para leer o modificar. Una de las complicaciones se presenta cuando existen copias de una relación, la BD replicada está en un estado consistente si todas las copias son idénticas.

El concepto de confiabilidad se refiere a la resistencia de un sistema ante distintos tipos de fallas y a su capacidad para recobrarse ante ellas. Un sistema resistente es tolerante a fallas del sistema mismo y puede continuar ofreciendo sus servicios una vez que se recupere. Un SMBD es recobrable si garantiza estados consistentes de la

BD después de haber ocurrido una falta, ya sea regresando a un estado consistente previo o cambiando a un nuevo estado consistente.

El manejo de transacciones se ocupa de mantener la BD en un estado consistente aun cuando ocurran accesos concurrentes o fallas.

CONCEPTO DE TRANSACCIÓN

Las actualizaciones de la BD se originan generalmente por eventos del mundo real como, por ejemplo, la recepción de un nuevo pedido en un sistema de ventas. Situaciones como estas usualmente involucran varias acciones, tales como:

1. Añadir el nuevo pedido a la tabla PEDIDOS.
2. Actualizar el total de ventas para el vendedor que promovió el pedido.
3. Actualizar el total de ventas para la oficina o departamento que atiende dichas ventas.
4. Actualizar las existencias para el producto solicitado.

Para que la BD esté en un estado consistente, todas las acciones necesarias deben ejecutarse como una unidad. En general una transacción es una secuencia de operaciones sobre la BD que deben ejecutarse atómicamente.

El SMBD es responsable de quese ejecuten todas las sentencias, o ninguna, si la aplicación aborta o se produce un fallo de hardware durante la ejecución de la transacción.

Las sentencias fundamentales son:

1. BEGIN TRANSACTION: inicio de transacción.
2. END TRANSACTION: fin del código de la transacción.
3. COMMIT: indica al SMBD terminación con éxito.
4. ROLLBACK: indica terminación sin éxito y deben desestimarse las acciones realizadas.

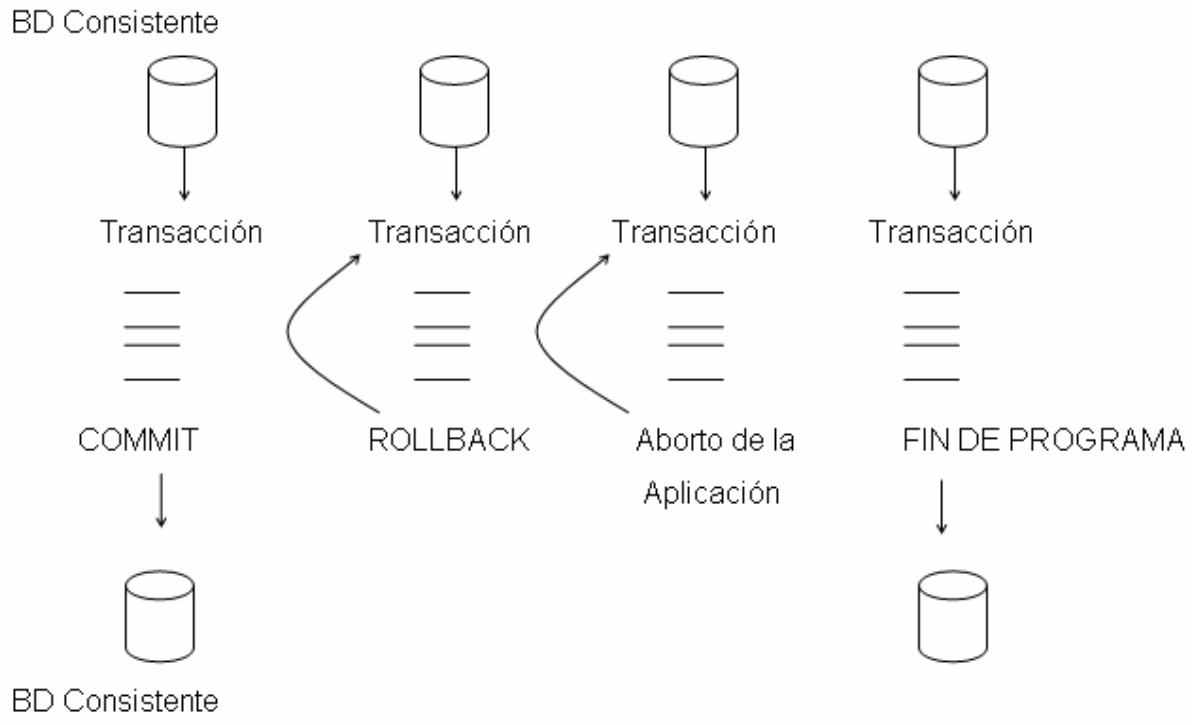
Ejemplo: Modificación de un pedido. Cambiar la cantidad solicitada del producto QSA - XK47 en el pedido número 113 051. Se desean 10 unidades con un importe de 3 550. En el pedido anterior había 4 unidades por un importe de 1 458. Este pedido lo atendió el empleado 108 de la oficina 21. Esta información se puede recuperar de los ya almacenados pero se obvian pasos para mayor claridad.

```
BEGIN TRANSACTION
UPDATE PEDIDOS
SET CANT = 10, IMPORTE = 3550
WHERE NR - PED = 113051;
UPDATE REPVENTAS
SET VENTAS = VENTAS - 1458 + 3550
WHERE NR - EMP = 108;
UPDATE OFICINAS
SET VENTAS = VENTAS - 1458 + 3550
WHERE NR - OFI = 21;
UPDATE PRODUCTOS
SET EXISTENCIAS = EXISTENCIAS - 6
WHERE ID - FAB = ‘QSA’ and ID - PROD = ‘XK47;
IF el usuario confirma este proceso
Then
COMMIT
Else
ROLLBACK
END - TRANSACTION;
```

El estándar SQL especifica que una transacción empieza de manera automática con la primera sentencia SQL ejecutable, la transacción continúa con las siguientes sentencias hasta finalizar en uno de los modos que aparecen a continuación:

1. Al encontrar una sentencia COMMIT. Termine con éxito la transacción actual y crea las condiciones para el inicio de una nueva transacción.
2. Al encontrar una sentencia ROLLBACK. Aborte la transacción actual y cree las condiciones para el inicio de una nueva transacción.
3. Al llegar al fin de programa, lo que equivale a la terminación con éxito de la transacción actual.
4. Al abortar una aplicación, lo que equivale a una terminación sin éxito.

Estos casos se pueden resumir en el gráfico siguiente:



Bajo el modelo ANSI/ISO el usuario o la aplicación siempre tienen definida una transacción. Las sentencias COMMIT y ROLLBACK también se pudieran usar en modo interactivo pero rara vez se usan en este contexto. Como que las actualizaciones multisentencias no son propias del modo interactivo, los productos SQL interactivos asumen un COMMIT después de cada sentencia pues de hecho,

cada sentencia es una transacción, lo cual equivale al cuarto de los modos mencionados.

Ejemplo: diseñar una transacción para un sistema de reservación de una aerolínea. Considerar una versión simplificada de una problemática típica en que se capta número de vuelo, nombre del pasajero, y si hay asientos disponibles se hace la reservación de uno de ellos.

Esquemas: Se denotan por P, Q y R respectivamente para facilitar la calificación.

P VUELOS (no.-vuelo, fecha, origen, destino, asientos-disp, capacidad)

Q PASAJERO (p-nombre, dirección)

R PAS-VUE (no - vuelo, fecha, p-nombre, clase)

```
BEGIN - TRANSACTION Reservación
Begin
Input (vuelo, día, pasajero);
EXEC SQL  SELECT asientos - disp, capacidad
          INTO Temp1, temp2
FROM VUELOS
WHERE nro-vuelo = vuelo AND Fecha = día;
If temp1 = 0 then
Begin
 Output ("No hay asientos disponibles")
 ROLL BACK
End
Else
Begin
          EXEC SQL  UPDATE VUELOS
                    SET asientos- disp = asientos - disp - 1
                    WHERE nro-vuelo = vuelo and fecha = día;
          EXEC SQL  INSERT INTO PAS-VUE
```

```
VALUES (Vuelo, día, pasajero, null);
                COMMIT
                Output ("reservación OK")
        End
End-if
End
```

Nóte en el ejemplo anterior que la notificación al usuario de la confirmación o no de su reservación se hace después del COMMIT si se confirma, o antes del ROLLBACK si no se confirma; por razones de seguridad.

CARACTERIZACIÓN DE TRANSACCIONES

Se debe observar en los ejemplos precedentes que las transacciones leen y escriben algunos datos. Los datos que una transacción lee constituyen su conjunto de lectura (RS),de manera similar los datos que una transacción escribe constituyen su conjunto de escritura (WS); la unión de ambos conjuntos (no necesariamente excluyentes) constituye su conjunto base (BS= RS U WS).

Ejemplo:

RS [Reservación] = { asientos - disp, capacidad }
WS [Reservación] = { asientos-disp, R. nro-vuelo, R.fecha, R.p-nombre, R.clase}

BS = RS U WS
Se omite P.nro-vuelo y P.fecha en RS para simplificar el ejemplo.

Las transacciones se caracterizan inicialmente sobre la base de operaciones SELECT Y UPDATE sin considerar INSERT y DELETE. O sea, operaciones sobre BD estáticas que no crecen ni decrecen. Las BD dinámicas tienen que tratar con otros problemas.

FORMALIZACIÓN DEL CONCEPTO DE TRANSACCIÓN

Denotar por O_{ij} (x) : alguna operación O_j de la transacción T_i sobre la entidad x.

$O_j \in \{$ SELECT, UPDATE $\}$, por sencillez se denotará $\{$ read, write $\}$

OS_i: Conjunto de todas las operaciones en T_i ($OSi = U_j O_{ij}$)

N_i: Condición de terminación de T_i, $N_i \in \{$ COMMIT, ROLLBACK $\}$

Una *transacción* T_i es un ordenamiento parcial sobre sus operaciones y la condición de terminación.

Un orden parcial $P = \{ \sum, \propto \}$ define un ordenamiento entre los elementos de $\sum$, llamado el dominio, acorde a una relación binaria $\propto$ irreflexiva y transitiva definida sobre $\sum$. En este caso $\sum$ consiste las operaciones y la condición de terminación de una transacción, mientras que $\propto$ indica el orden de ejecución de estas operaciones.

Formalmente, una transacción es un orden parcial.

$T_i = \{ \sum_i, \propto_i \}$, donde:

1. $\sum_i = OS_i \cup \{ N_i \}$
2. Para dos operaciones cualesquiera O_{ij}, $O_{ik} \in OS_i$, si O_{ij} = R (x) y O_{ik} = W (x) para algún elemento de dato x, entonces $O_{ij} \propto_i O_{ik}$ o $O_{ik} \propto_i O_{ij}$
3. $\forall O_{ij} \in OS_i$, $O_{ij} \propto_i N_i$

La condición 1 define el dominio como un conjunto de operaciones read, write y la condición de terminación; la condición 2 especifica la relación de orden entre

operaciones conflictivas mientras que la condición 3 indica que la condición de terminación siempre sigue a las demás.

La relación de orden es dependiente de la aplicación. Dos operaciones O_i (x) y O_j (x) están en conflicto si operan sobre el mismo dato y al menos una de ella es de escritura.

Ejemplo: Considere la transacción T como sigue:

Read (x)
Read (y)
X: = x + y
Write (x)
Commit

Esta transacción se define como:

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

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

Donde: Un elemento (O_i, O_j) de la relación $\propto$ indica que $O_i \propto O_j$

Nótese que la relación de orden no especifica el orden relativo de todas las operaciones respecto a la condición de terminación, tampoco se especifica el orden entre todo par de operaciones pues es un orden parcial.

Ejemplo: la transacción Reservación. Observar que hay dos posibles condiciones de terminación en dependencia de la disponibilidad de asientos (plazas). La definición de transacción indica que solo hay una condición de terminación pero debe aclararse

que este concepto se asocia al de ejecución. O sea, hay dos transacciones definidas, una que aborta y otra exitosa.

T_1: (Transacción que se aborta)

$\sum = \{ R (asientos - disp), R (capacidad), R\} = \{O_1, O_2, R\}$

$\propto = \{ (0_1, R), (0_2, R) \}$

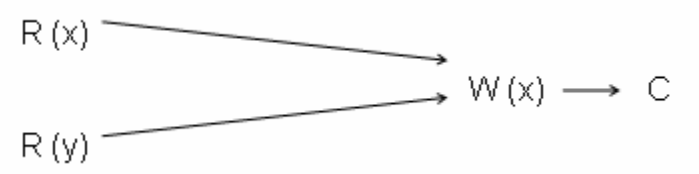
T_2: (Transacción que se compromete)

$\sum = \{R (asientos - disp), R (capacidad), W (asientos-disp), W (nro - vuelo),$
$W (fecha), W (p-nombre), W (clase), C \}$

$\propto = \{ (0_1, 0_3), (0_2, 0_3), (0_1, 0_4), (0_1, 0_5), (0_1, 0_6), (0_1, 0_7), (0_2, 0_4), (0_2,0_5),$
$(0_2, 0_6), (0_2, 0_7), (0_1, C), (0_2, C), (0_3, C), (0_4, C) (0_5, C), (0_6, C)$
$(0_7, C) \}$

Las operaciones O_i se corresponden en orden con las definidas en $\sum$.
Una ventaja de definir una transacción como un orden parcial es su correspondencia con un gráfico acíclico dirigido (DAG). Luego una transacción puede ser especificada como un DAG cuyos vértices son las operaciones de una transacción y cuyos arcos indican las interrelaciones de orden entre un par dado de operaciones. Esto es útil para el resto de la teoría del manejo de transacciones con teoría de grafos.

Ejemplo de la transacción T ya descrita:



No se representan arcos que se obtienen por transitividad. Generalmente no se necesita una referencia por separado al dominio del orden parcial y a la relación de

orden; luego, es frecuente usar el nombre de la relación de orden parcial para referirse a ambos. Es conveniente especificar el orden de las operaciones de una transacción haciendo uso de su orden relativo en la definición de la transacción.

Ejemplo: $T = \{ R(x), R(y), W(x), C \}$

PROPIEDADES DE LAS TRANSACCIONES

Se ha descrito el concepto de transacción, sin embargo, no se ha justificado por qué es una unidad de consistencia y cómputo confiable; estos aspectos se deben a cuatro propiedades, conocidas como propiedades ACID:

1. Atomicidad
2. Consistencia
3. Aislamiento
4. Durabilidad

A continuación se hace un análisis de cada una de ellas:

1. Atomicidad: se refiere al hecho que una transacción es tratada como una unidad de operación, o sea, todas las acciones de una transacción se completan o ninguna, lo que se conoce como “Todo o nada“. Si una transacción se interrumpe debido a una falla, hay dos posibles formas de tratar el problema: la transacción puede ser terminada posteriormente completando las acciones que faltan o puede ser terminada deshaciendo las acciones ya ejecutadas. Se presentan generalmente dos tipos de fallas.

1.1. Una transacción puede fallar debido a errores en datos de entrada, deadlock u otros factores. En estos casos la propia transacción o el SMBD emite el ROLLBACK: mantener atomicidad en la presencia de este tipo de falla se conoce como recobrado (*recovery*).

1.2. Una transacción puede fallar por “desastres (*crash*)” en el sistema tales como fallas del procesador, falta de fluido eléctrico, etc. Para asegurar atomicidad en la presencia de este tipo de falla se dispone del *crash recovery*.

2. Consistencia: La consistencia de una transacción es su corrección, una transacción es un programa correcto que transforma un estado consistente de la BD en otro. Verificar que las transacciones sean consistentes es tarea del control de datos semánticos; asegurar consistencia es el objetivo de los mecanismos de control de concurrencia. Hay una clasificación de consistencia que establece cuatro niveles, el concepto de dato sucio se refiere a valores que han sido escritos por una transacción antes de comprometerse.

Grado 3 de consistencia para una transacción T si:

G3.1. T no sobreescribe datos sucios de otras transacciones.
G3.2. T no compromete una escritura hasta que se compromete completa.
G3.3. T no lee datos sucios de otras transacciones.
G3.4. Otras transacciones no ensucian datos leídos por T antes que ésta se comprometa.

Grado 2 de consistencia para una transacción T si:

G2.1. T no sobreescribe datos sucios de otras transacciones.
G2.2. T no compromete escritura antes de terminar.
G2.3. T no lee datos sucios de otras transacciones.

Grado 1 de consistencia para una transacción T si:

G1.1. T no sobreescribe datos sucios de otras transacciones.
G1.2. T no compromete escritura antes de terminar.

Grado 0 de consistencia para una transacción T si:

G0.1. T no sobreescribe datos sucios de otras transacciones.

3. Aislamiento: Es la propiedad que plantea que cada transacción tenga acceso a una BD consistente todo el tiempo, o sea, una transacción activa no puede revelar sus resultados a otras transacciones concurrentes antes de comprometerse.

Ejemplo: Considerar las dos transacciones concurrentes siguientes que acceden al mismo dato x. Asumir x = 50

T_1: Read (x)	T_2: Read (x)
x:= x + 1	x : = x + 1
Write (x)	Write (x)
Commit	Commit

La siguiente es una posible secuencia de ejecución de estas transacciones:

T_1:	Read (x)	x = 50
T_1:	x: = x+ 1	
T_1:	write (x)	
T_1:	commit	
T_2:	Read (x)	x = 51
T_2:	x: = x + 1	
T_2:	write (x)	
T_2:	commit	

La anterior ejecución es correcta, no obstante, si hay concurrencia también es posible la ejecución siguiente:

T_1:	Read (x)	x = 50

T_1: x : = x + 1

T_2: Read (x) x = 50

T_1: Write (x)

T_2: x : = x + 1

T_2: write (x)

T_1: commit

T_2: commit

Resultado incorrecto, T_2 lee x mientras su valor está siendo cambiado. Asegurar aislamiento al no permitir que resultados incompletos sean accedidos por otras transacciones resuelve el problema anterior.

Otra razón para el aislamiento es el aborto en cascada, o sea, si una transacción accede a un resultado no comprometido y la transacción que originó ese resultado decide abortar, también debe hacerlo la otra. Esta cadena ocasiona sobrecarga al sistema.

También es posible establecer niveles de consistencia desde la perspectiva de la propiedad de aislamiento, en la medida que se sube en la jerarquía de consistencia hay mayor aislamiento. Grado 0 evita actualización perdida; Grado 2 evita aborto en cascada, y Grado 3 proporciona aislamiento completo y obliga a una transacción conflictiva a esperar por otra.

4. Durabilidad: se refiere a la propiedad que asegura que cuando una transacción se compromete, sus resultados se hacen permanentes. Se ocupa de asegurar que la BD se lleve a un estado consistente cuando se reflejan todas las acciones comprometidas.

TIPOS DE TRANSACCIONES

Aunque los problemas que se presentan en cada tipo son similares, existen diferencias en los algoritmos y técnicas para resolverlos. Se pueden clasificar las transacciones de acuerdo a un número de criterios, como por ejemplo:

1. Por áreas de aplicación: pueden ser transacciones centralizadas, transacciones distribuidas, transacciones heterogéneas (si el ambiente de BD es heterogéneo). Implícitamente se designa por transacción cuando trabaja sobre un ambiente centralizado y homogéneo.
2. Por su duración: transacciones *on-line* o en lotes (*batch*), también denominadas de corta vida o larga vida. Las primeras necesitan tiempos de respuestas cortos y usualmente acceden a porciones pequeñas de la BD. Las transacciones en *batch* se caracterizan por lo contrario. Por ejemplo, aplicaciones CAD.
3. Por su estructura: transacciones simples o anidadas, las que tienen algunas restricciones.

Si se atiende la estructura de las transacciones acorde a cómo sus acciones read o write son organizadas, entonces se pueden clasificar en otras formas. Los ejemplos que se han mostrado mezclan estas sentencias sin un orden específico. A estas transacciones se les puede llamar transacciones generales. Si existe la restricción de que todas las acciones de lectura se ejecuten antes de las acciones de escritura, se llaman transacciones en dos pasos. Si existe la restricción de que un elemento de dato tenga que ser leído antes de su actualización (escrito), se llaman transacciones restringidas o de lectura antes de escritura. Una transacción puede tener las dos restricciones anteriores y recibe la combinación de ambos nombres.

Existe un modelo de transacciones que consiste en la clase restringida con la restricción adicional que cada <read, write> debe ejecutarse atómicamente, se conoce como “acción”.

Ejemplo:

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(z), W(y), W(w), C}

Restringida:

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

Restringido en dos pasos:

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

Acción:

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

Estos tipos de transacciones requieren tratamientos diferentes.

ARQUITECTURA DEL MONITOR DE EJECUCIÓN DE TRANSACCIONES

El monitor de ejecución distribuida consiste en dos módulos: un manejador de transacciones (TM), un planificador (SC, de *scheduler*) y un manejador de recobrado local (LR)

TM: Coordina la ejecución de operaciones

SC: Instrumenta los algoritmos de control de concurrencia para sincronizar los accesos a la BD.

LR: Instrumenta los procedimientos locales para recuperar las BD locales a un estado consistente después de una falla.

El sitio donde se origina una transacción se conoce como su sitio de origen, el TM de ese sitio coordina las operaciones de las transacciones que se originan en el mismo. El TM instrumenta una interfaz para los programadores de aplicación que consiste en cinco comandos.

1. Begin transaction: indica al TM que se inicia una nueva transacción y éste controla un conjunto de información sobre su ejecución.
2. Read: si el elemento de dato leído x se almacena localmente, su valor es leído y retorna a la transacción, si no, se procede de manera similar buscando en una copia de la BD que contiene x.
3. Write: TM coordina la actualización de los valores de x en cada sitio donde residen sus copias.
4. Commit: TM coordina la actualización física de todas las copias que contienen los datos actualizados.
5. Rollback: TM asegura que ningún efecto de la transacción se refleje en la BD.

Por simplicidad, se ha ignorado la planificación de transacciones concurrentes así como los detalles de la recuperación física de los datos. Para proporcionar los servicios descritos, los TMs deben comunicarse con los SCs y los procesadores de datos de sus mismos sitios o de otros.

CAPÍTULO 3. CONTROL DE CONCURRENCIA

El control de concurrencia trata con las propiedades de aislamiento y consistencia de las transacciones. Los mecanismos de control de concurrencia distribuido de un SMBDD aseguran la consistencia de la BD en un ambiente distribuido. Si las transacciones son internamente consistentes, la forma más fácil de garantizar una consistencia intertransacciones es ejecutando una después de la otra. De manera obvia esto no es aceptado pues limita el uso racional de los recursos. Uno de los parámetros más importantes que caracterizan un sistema distribuido es la cantidad posible de transacciones concurrentes y deben atenderse bien los mecanismos que controlen su interacción. Se acostumbra comenzar el estudio del control de concurrencia a partir de la teoría de serialización, siendo la serialización el criterio de correctividad más aceptado para los algoritmos de control de concurrencia.

Hay tres problemas importantes que pueden surgir si las transacciones no se manejan adecuadamente.

1. El problema de la actualización perdida: suponer que las transacciones siguientes acuden concurrentemente a una tabla sobre Productos con los atributos: código del fabricante, código del producto y existencias.

T_1 T_2

12:00 PROD

ID-FAB	ID-PROD	EXIST
ACT	41004	139

12:01

SELECT EXIST

FROM PROD

WHERE id-fab= “ACT” and

id-prod = 41004

----------139----------

12:02
SELECT EXIST
FROM PROD
WHER id-fab=“ACT”and id-prod=41004
----------139----------

12:03 Acepta un pedido de 125

12:04 Acepta un pedido de 100

12:05
UPDATE PROD
SET EXIST=39

12:06
UPDATE PROD
SET EXIST = 14

La gestión de los dos pedidos ha originado errores, ambos pedidos se aceptaron cuando habían 139 unidades del producto seleccionado pero no habían suficientes para las dos. Se presentó el problema conocido por “actualización perdida” y se caracteriza por la lectura de los mismos datos y su modificación posterior.

2. El problema de los datos no comprometidos: sobre la misma información; proceder de la manera siguiente:

T_1 T_2

12:00 PROD

ID-FAB	ID-PROD	EXIST
ACT	41004	139

12:01
SELECT EXIST
FROM PROD
 WHERE id-fab= “ACT” and
 id-prod = 41004

----------139----------

12:03 Acepta un pedido de 100

12:04

UPDATE PROD

SET EXIST=39

ID-FAB	ID-PROD	EXIST
ACT	41004	39

12:05

SELECT EXIST

FROM PROD

WHER id-fab="ACT"and id-prod=41004

----------39----------

Llega un pedido de 125 y se rechaza. Se solicitan más productos al proveedor.

12:06 ROLLBACK

El problema se presenta porque T_2 consultó una actualización no comprometida.

3. El problema de los datos inconsistentes: para mostrar este problema suponer que se desean calcular los totales de los pedidos de un determinado cliente mediante una transacción T_1 y que otra transacción T_2 se encarga de hacer actualizaciones en la tabla correspondiente a los pedidos, esta tabla contiene el número del pedido, el código del cliente y el importe correspondiente. Un pedido se puede cancelar, modificar o añadir uno nuevo.

T_1 T_2

11:29 X 31500

11:30 Y 3745

12:00

PEDIDOS …

12:00 SELECT * FROM PEDIDOS

	11:29 X 31500

	11:30 Y 3745
	Sum:=Suma de un pedido de Y.
12:03	
DELETE FROM PEDIDOS	
WHERE nro-pedido=1130	
12:04	
INSERT INTO PEDIDOS	
VALUES (1131, Y ,5000)	
12:05 COMMIT	
	12:06
	Sum:= suma iterada de otro pedido de Y
	Sum:= 3745 + 5000
	…
	COMMIT

Observar que T_2 tuvo acceso a datos antes y después de hacer modificaciones.

Como muestran los ejemplos anteriores, cuando varios usuarios comparten una BD, existe una posibilidad de corrupción de los datos, aun cuando cada transacción se ejecute correctamente. Para el estudio de los mecanismos que manejan la ejecución concurrente de transacciones se asume ausencia de fallas.

El criterio de corrección más ampliamente usado para el control de concurrencia es el de seriabilidad. Se debe insistir en que cuando varias transacciones se ejecutan de manera concurrente, la consistencia de la BD se puede perder, aun cuando cada transacción sea correcta. El concepto de seriabilidad ayuda a identificar las ejecuciones concurrentes que aseguran la consistencia.

TEORÍA DE LA SERIABILIDAD

Una planificación, también conocida por historia, se define sobre un conjunto de transacciones $T=\{T_1,\ldots,T_n\}$ y especifica un orden de los entrecruzamientos de las sentencias de dichas transacciones. Sobre la base de la definición ya dada para una transacción, una planificación puede ser definida como un orden parcial sobre T. Antes de dar una definición formal se enfocan algunos aspectos preliminares. Se debe recordar que dos operaciones $O_{ij}(x)$ y $O_{kl}(x)$ (i, k no necesariamente distintos) que acceden al mismo elemento de dato se son conflictivas si al menos una de ellas es de escritura. Observar que dos operaciones de lectura sobre el mismo dato, no entran en conflicto. Hay, por tanto, dos tipos de conflictos: lectura-escritura (o escritura-lectura) y escritura-escritura.

Las dos operaciones pueden pertenecer a la misma transacción o a diferentes transacciones, en cuyo caso se dice que las transacciones son conflictivas. La existencia de conflicto entre dos operaciones indica que su orden de ejecución es importante.

Para la comprensión de esta teoría, véase el planteamiento informal siguiente. Por comodidad se asume que las operaciones read (Q, q) y write (Q, q) son equivalentes a read (Q) y write (Q) respectivamente; o sea, que la variable temporal tiene igual nombre que la del dato a que se accede. Considerar las transacciones siguientes.

T_0- transfiere fondos de una cuenta A a otra B, suponer que A =1000 y B =2000 (representan los montos de los fondos).

T_0: read (A)
A:=A-50
Write (A)
read (B)
B:=B+50

write (B)

T_1 - Transfiere el 10 % del saldo de la cuenta A a la B

T_1: read (A)
temp:= A*0.1
A:= A- temp
write (A)
read (B)
B:=B + temp
write (B)

Suponer que las transacciones se ejecutan en serie T_o-T_1, los resultados son:

Para T_0: Saldo de A =950, Saldo de B= 2 050
Para T_1: Saldo de A =950 - 95= 855, Saldo de B = 2 050+ 95=2 145
Saldo total de A +B =3000

Si se ejecutan en serie T_1-T_o, los resultados son:
Para T_1: Saldo de A=1 000-100 =900, Saldo de B=2 000+ 100=2 100
Para T_0: Saldo de A=900 – 50 = 850, Saldo de B =2100+ 50=2 150
Saldo total de A +B=3 000

Ambas ejecuciones conservan la consistencia de la BD.

Las secuencias de ejecución descritas representan el orden cronológico en que se ejecutan las sentencias y reciben el nombre de planificaciones. Sea P_1 el orden T_0-T_1 y P_2 el orden T_1-T_0.

Planificación en serie: cuando las sentencias de una misma transacción aparecen todas juntas y en el mismo orden. Ejemplo: P_1 y P_2.

Para n transacciones hay n! planificaciones en serie. Cuando se ejecutan transacciones concurrentes no es preciso que las planificaciones estén en serie, entonces la cantidad posible de planificaciones es mucho mayor que n!. Observe la posibilidad siguientes:

P_3: T_0	T_1
read (A)	
A:=A-50	
write (A) (950)	
	Read (A)
	temp:=A*0.1
	A:=A-temp
	write (A) (855)
read (B)	
B:=B+50	
write (B) (2050)	
	read (B)
	B:=B+temp
	write (B) (2145)

Al concluir P_3 el saldo de A es 855 y el de B es 2 145 por lo que se conserva la suma total de 3 000. Este resultado coincide con el ya descrito para P_1. A este tipo de planificación, como P_3 que es equivalente a una en serie o serial, se le llama seriabilizable.

Ejemplo de planificación concurrente no seriabilizable:

P_4: T_0	T_1
read (A) (1000)	
A: = A-50 (950)	
	read (A) (1000)
	temp:= A*0.1 (100)

	A:=A-temp	(900)
	write (A)	(900)
	read (B)	(2 000)
write (A) (950)		
read (B) (2 000)		
B:=B+50 (2 050)		
write (B) (2 050)		
	B:=B+temp	(2 100)
	write (B)	(2100)

Como puede observarse, los últimos valores para A y B fueron de 950 y 2100 respectivamente, con un saldo total de 3 050.

Las transacciones son programas, es difícil determinar cómo interactúan sus operaciones, por eso se simplifica el análisis con frecuencia solo a operaciones read y write.

Formalizando estos conceptos. Una planificación completa define el orden de ejecución de todas las operaciones en su dominio, una planificación se define como un prefijo de una planificación completa. Una planificación completa S_T^c definida sobre un conjunto de transacciones T= {T_1,….T_n} es un orden parcial $S_T^c = \{\sum_T , \propto_T\}$donde:

1. $\sum_T = \bigcup_{i=1}^{n} \sum_i$

2. $\propto_T \supseteq \bigcup_{i=1}^{n} \propto_i$

3. Para dos operaciones conflictivas cualesquiera O_{ij}, $O_{kl} \in \sum_T$, $O_{ij} \propto_T O_{kl}$ ó $O_{kl} \propto_T O_{ij}$.

La primera condición plantea que el dominio de la planificación es la unión de los dominios de transacciones individuales; la segunda condición define la relación de

orden como un superconjunto de las relaciones de orden de las transacciones individuales (éste mantiene el ordenamiento de operaciones dentro de cada transacción) y la condición final define el orden de ejecución entre operaciones conflictivas.

Ejemplo: Considerar dos transacciones T_1 y T_2

T_1:	T_2:
Read (x)	Read (x)
x:=x+1	x:=x+1
write (x)	write (x)
commit	commit

Una posible planificación completa S_T^c sobre T= {T_1, T2} puede ser expresado como el orden parcial siguiente (los subíndices indican la transacción).

$$S_T^c = \left\{ \sum_T \ , \propto T \right\}$$

Donde: $\sum_1$={$R_1(x)$, W, (x), C_1} y $\sum_2$={ $R_2(x)$, $W_2(x)$, C_2}

Luego: $\sum_T = \sum_1 \cup \sum_2$={$R_1(x)$, $W_1(x)$, C_1, $R_2(x)$, $W_2(x)$, C_2 } y

$\propto_T$={(R_1,R_2), (R1,W1), (R_1,C_1), (R_1, W_2), (R_1,C_2), (R_2, W_1), (R_2, C_1), (R_2,W_2)
(R_2, C_2), (R_2, W_2), (R_2, C_2), (W_1, C_1), (W_1, W_2), (W_1, C_2), (C_1, W_2),
(C_1, C_2), (W_2, C_2)}

Estas transacciones pueden ser especificadas como un grafo dirigido acíclico (DAG). Observe que no se han representado los arcos implicados por transitividad. Para simplificar la notación de la planificación, frecuentemente se especifica como una lista de las operaciones en $\sum_T$, luego S_T^c={$R_1(x)$, $R_2(x)$, $W_1(x)$,C_1, $W_2(x)$, C_2}

Una planificación se define como un prefijo de una planificación completa. Dado un orden parcial P= {$\sum$, $\propto$}, P'= {$\sum$', $\propto$'} es un prefijo de P, si:

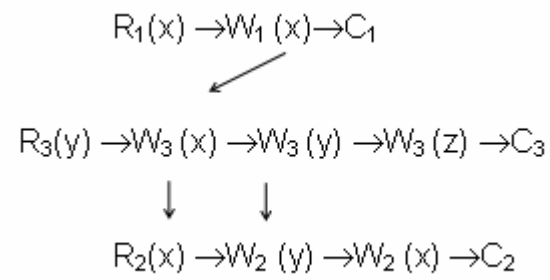
1. $\Sigma' \subseteq \Sigma$
2. $\forall e_i \in \Sigma'$, $e_1 \propto' e_2$ si y solo si $e_1 \propto e_2$
3. $\forall e_i \in \Sigma'$, si $\exists\ e_j \in \Sigma$ tal que $e_j \propto e_i$ entonces $e_j \in \Sigma'$

Las dos primeras condiciones definen P' como una restricción de P sobre Σ', luego las relaciones de orden en P se mantienen en P'. La última condición indica que para cualquier elemento de Σ' todos sus predecesores en Σ se deben incluir en Σ'.

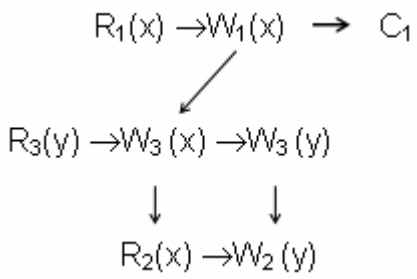
Ejemplo:

$T_1 = \{R_1(x), W_1(x), C_1\}$
$T_2 = \{R_2(x), W_2(y), W_2(x), C_2\}$
$T_3 = \{R_3(y), W_3(x), W_3(y), W_3(z), C_3\}$

Una planificación completa sobre T_1-T_2-T_3 es:



Una planificación prefijo de lo anterior, es:



Este tipo de planificaciones incompletas son importantes debido a la posibilidad de ocurrencia de fallas. Si en una planificación S, las operaciones de las distintas transacciones no son mezcladas, se dice que la planificación es serial. Como ya se

describió, una ejecución serial de un conjunto de transacciones mantiene la consistencia de la BD, lo que se justifica naturalmente de la propiedad de consistencia de cada transacción por separado.

Ejemplo:

$S=\{R_2(x), W_2(y), W_2(x), C_2, R_1(x), W_1(x), C_1, R_3(y), W_3(x), W_3(y), W_3(z), C_3\}$

Esta ejecución corresponde al orden T_2-T_1-T_3.

Basadas en las interrelaciones de precedencia introducidas por el orden parcial, es posible discutir la equivalencia de planificaciones respecto a sus efectos sobre la BD.

Intuitivamente, dos planificaciones S_1 y S_2 definidas sobre el mismo conjunto de transacciones T, son equivalentes si tienen el mismo efecto sobre la BD. Formalmente, dos planificaciones S_1 y S_2 definidas sobre el mismo conjunto de transacciones T, se dice que son equivalentes si para cada par de operaciones conflictivas O_{ij} y O_{kl} (i distinto de k), si $O_{ij} \propto_1 O_{kl}$, entonces $O_{ij} \propto_2 O_{kl}$. A este tipo de equivalencia se le llama equivalencia en conflicto debido a que define equivalencia de dos planificaciones en términos del orden relativo de ejecución de operaciones en conflicto. Se asumen transacciones no abortadas.

Ejemplo:

T_1	T_2	T_3
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

Sean $S_1=\{ W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}$

$S_2=\{W_2(x), R_1(x), R_3(x), W_1(x), C_1, W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3\}$

¿S_1 y S_2 son equivalentes en conflictos? Observar que se invirtió el orden de dos operaciones en conflicto.

Una planificación es seriabilizable si y sólo si es equivalente en conflicto a una planificación en serie. Nóte su similaridad con la consistencia de grado 3 ya descrita.

La teoría de la seriabilidad es también conocida como seriabilidad basada en conflicto. El conflicto en planificaciones seriabilizables se puede definir de la manera siguiente:
Sea S una planificación y dos instrucciones I_i, I_j de las transacciones T_i, T_j, respectivamente.

1. Si I_i, I_j se refieren a diferentes datos, se pueden intercambiar.
2. Si I_i, I_j se refieren al mismo dato, puede ser que interese el orden en que se ejecutan. Hay cuatro casos:
2.1. I_i=read(Q) I_j=read(Q) no interesa el orden.
2.2. I_i=read(Q) I_j=write(Q) interesa el orden pues no se obtienen los mismos resultados si se conmutan las operaciones, por ejemplo en el orden I_i-I_j no se lee el dato que se acaba de escribir.
2.3. I_i=write(Q) I_j=read(Q) similar al anterior
2.4. I_i=write(Q) I_j=write(Q) el orden de las operaciones define el último valor almacenado en la BD.

Como se puede observar, dos operaciones están en conflicto si son operaciones de transacciones diferentes, sobre el mismo dato y por lo menos una de ellas es la escritura.
Sea P_5:

T_0	T_1
Read(A)	
Write(A)	

	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

P_5 es equivalente a T_0-T_1 y, por tanto, es seriabilizable. Instrucciones no conflictivas se pueden intercambiar para obtener una nueva planificación seriabilizable.

Ejemplo:

T_0	T_1
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

Una planificación es seriabilizable en conflictos si es equivalente en conflictos a una planificación en serie y viceversa.

Sea P_6:

T_2	T_3
Read(Q)	
	Write(Q)

Write(Q)

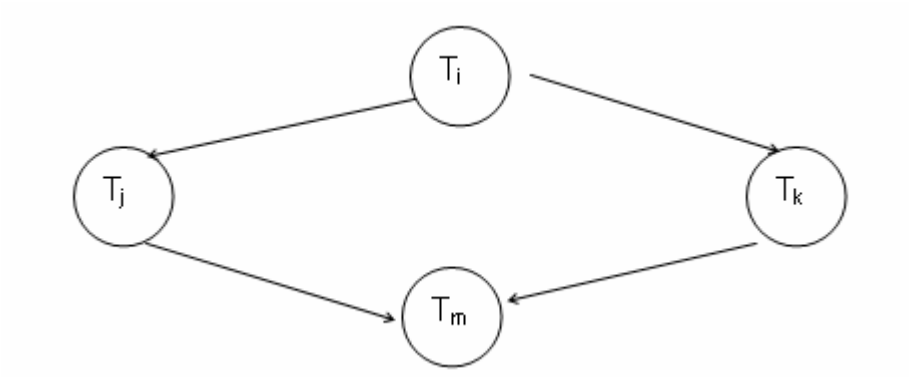
¿Es seriabilizable? Observar que no es equivalente ni a T_2-T_3 ni a T_3-T_2.

Existe otro tipo de equivalencia conocido como equivalencia de vistas que no se aborda en este caso, pues la equivalencia en conflictos es más útil para el control de concurrencia. La principal función del controlador de concurrencia es generar planificaciones seriabilizables. La teoría de la seriabilidad se extiende de manera directa a las BD no replicadas (o particionadas).

Suponer que se quiere probar si es seriabilizable en conflicto, para ello se dispone de la prueba de seriabilidad siguiente.

Sea S una planificación:
1. Construir un grafo dirigido llamado grafo de precedencia de S de la forma siguiente:
G=(V,E)
V: Conjunto de vértices que representan a todas las transacciones que participan en la planificación.
E: Conjunto de aristas T_i -> T_j que reflejan una de las condiciones siguientes:
1.1. T_i ejecuta write(Q) antes que T_j ejecute read(Q)
1.2. T_i ejecuta read(Q) antes que T_j ejecute write(Q)
1.3. T_i ejecuta write(Q) antes que T_j ejecute write(Q)
Si existe una arista $T_i \rightarrow T_j$ en el grafo de precedencia, esto implica que T_i debe aparecer antes que T_j en cualquier planificación en serie S'. Si el grafo de precedencia tiene un ciclo entonces la planificación S no es seriabilizable en conflictos, de lo contrario lo es. Un orden lineal de ejecución puede obtenerse mediante una ordenación topológica que determina varias posibilidades lineales correspondientes a un mismo orden parcial.

Ejemplo:



Tiene dos posibles ordenamientos: T_i-T_j-T_k-T_m y T_i-T_k-T_j-T_m. Esto nos indica que no existe ninguna planificación, por ejemplo, equivalente a T_k-T_m-T_i-T_j (ni ella misma).

La teoría de seriabilidad se extiende a BD no replicadas (particionadas), de manera directa. La planificación de la ejecución de una transacción en un sitio es llamada planificación local. Si la BD no es replicada, y cada planificación local es seriabilizable, su unión es también seriabilizable siempre que los órdenes de seriabilidades locales sean idénticos.

En una BDD replicada esta extensión requiere más cuidado pues es posible que las planificaciones locales sean seriabilizables pero aún la consistencia de la BD está comprometida. Como ejemplo considérense dos sitios y un elemento de datos x duplicado en ambos. Considerar además las transacciones siguientes:

T_1	T_2
read(x)	read(x)
X:=x+5	X:=x*10
Write(x)	Write(x)
Commit	Commit

Ambas transacciones necesitan ejecutarse en ambos sitios. Las siguientes dos planificaciones pueden ser generadas localmente en los dos sitios:

$S_1 = \{ R_1(X), W_1(X), C_1, R_2(X), W_2(X), C_2\}$
$S_2 = \{ R_2(X), W_2(X), C_2, R_1(X), W_1(X), C_1\}$

Notar que ambas son planificaciones en serie, por consiguiente son correctas, pero observar que ordenar a T_1 y T_2 en orden inverso. Suponer que x = 1 antes de la ejecución de estas transacciones.

Al finalizar S_1 en el sitio 1, x = 60
Al finalizar S_2 en el sitio 2, x = 15

Esto viola la consistencia mutua de esos datos locales. La consistencia mutua requiere que todos los valores de todos los datos replicados sean idénticos. Las planificaciones que mantienen consistencia mutua se denominan de una copia; las planificaciones globales seriabilizables de una copia deben cumplir las condiciones siguientes:

1. Cada planificación local debe ser seriabilizable.
2. Dos operaciones en conflicto deben tener el mismo orden relativo en todas las planificaciones locales en que aparezcan juntas.

La segunda condición asegura que el orden de seriabilidad sea el mismo en todos los sitios donde las transacciones conflictivas se ejecutan juntas. Se debe resaltar que los algoritmos de control de concurrencia aseguran seriabilidad sincronizando accesos conflictivos a la base de datos. En BD replicadas la tarea adicional de asegurar seriabilidad de una copia es usualmente responsabilidad del protocolo de control de réplicas.

Los mecanismos para el control de concurrencia se pueden clasificar atendiendo a varios criterios; en particular se pueden clasificar como métodos de control de concurrencia pesimistas y métodos de control de concurrencia optimistas.

1. Métodos pesimistas: Sincronizan la ejecución concurrente de transacciones antes de la ejecución de su ciclo de vida.
2. Métodos optimistas: Tratan la sincronización de las transacciones en el momento de su terminación.

Para ambas se dispone de algoritmos basados en bloqueo, en hora de entrada o híbridos. Las variantes basadas en bloqueo alcanzan sincronización entre transacciones empleando bloqueos físicos o lógicos sobre alguna porción de la BD llamada su granularidad, el tamaño de estas porciones es importante y constituye la granularidad del bloqueo.

También se puede tener una subdivisión adicional de acuerdo al manejo del bloqueo:

1. Bloqueo centralizado: Uno de los sitios de la red se designa como sitio primario donde se almacenan los bloqueos.
2. Bloqueo de copia primaria: Una de las copias, si hay varias, de cada unidad de bloqueo es designada como copia primaria, y es ésta la copia que se bloquea para acceder a una unidad en particular. El mecanismo de bloqueo de copia primaria distribuye la responsabilidad del manejo de los bloqueos entre los sitios.
3. Bloqueo descentralizado: El manejo de bloqueos se comparte entre los sitios. La ejecución de una transacción requiere de la participación y coordinación de planificadores en más de un sitio. Cada planificador local es responsable de las unidades de bloqueo en tal sitio; de esta forma el bloqueo al elemento de datos x se extiende a todos los sitios que contienen a x.

Los ordenamientos por tiempo (por hora de entrada) organizan el orden de ejecución de transacciones asignando hora de entrada tanto a las transacciones como a los elementos de datos que se almacenan en la BD. Estos dos algoritmos pueden ser: básicos, multiversión o conservativos. Algunos algoritmos basados en bloqueo

también usan hora de entrada para manejar su eficiencia y nivel de concurrencia, estos son algoritmos híbridos.

PROTOCOLOS BASADOS EN BLOQUEO

Una forma de asegurar la seriabilidad es exigir que el acceso a los datos se haga de manera mutuamente excluyente, es decir, mientras una transacción accede a un dato, ninguna otra puede modificarlo. El método más común que se utiliza para instrumentar lo anterior es permitir que una transacción acceda a un dato sólo si tiene un bloqueo sobre el dato.

Existen varios modos de bloqueo, los básicos son dos:
1. Compartido: Si una transacción T_i ha obtenido un bloqueo compartido (S) sobre Q, T_i puede leerlo pero no modificarlo.
2. Exclusivo: Si una transacción T_i ha obtenido un bloqueo exclusivo (X) sobre Q, T_i puede leerlo y modificarlo.

Matriz COMP (de compatibilidad de bloqueo)

	S	X
S	true	false
X	false	false

Las sentencias siguientes representan acciones de bloqueo:

Lock-S(Q): Solicitud de un bloqueo compartido sobre Q.
Lock-X(Q): Similar para un bloqueo exclusivo.
Unlock(Q): Desbloquear un dato Q.

Para acceder a un dato, la transacción primero debe bloquearlo. Si el dato ya está bloqueado por otra transacción con un modo incompatible debe esperar hasta que se desbloquee.

Una transacción debe bloquear un dato mientras accede a él y no es siempre deseable que se desbloquee de inmediato después de su último acceso pues puede no garantizarse la seriabilidad.

Ejemplo:

T_1: transfiere $ 50.00 de la cuenta B a la A.

Lock-X(B)
Read(B)
B:=B-50
Write(B)
Unlock(B)
Lock-X(A)
Read(A)
A:=A+50
Write(A)
Unlock(A)

T_2: presenta el monto total de las cuentas A y B.

Lock-S(A)
Read(A)
Unlock(A)
Lock-S(B)
Read(B)
Unlock(B)
Display(A+B)

Suponer que los montos de las cuentas A y B son 100 y 200, respectivamente. Tanto la planificación T_1-T_2 como T_2-T_1 muestran 300 unidades y hacen la transferencia correctamente. Sin embargo, si se ejecuta de manera concurrente pueden presentarse inconsistencias debido a desbloqueos demasiado rápidos.

Ejemplo:

P_7: T_1	T_2
Lock-X(B)	
Read(B) (200)	
B:=B-50	
Write(B) (150)	
Unlock(B)	
	Lock-S(A)
	Read(A) (100)
	Unlock(A)
	Lock-S(B)
	Read(B) (150)
	Unlock(B)
	Display(A+B) (250)
	Resultado erróneo pues se desbloqueó B muy rápido
Lock-X(A)	
Read(A)	
A:=A+50	
Write(A)	
Unlock(A)	

Suponer las mismas transacciones pero con los bloqueos retrasados:

T_1	T_2
Lock-X(B)	Lock-S(A)
Read(B)	Read(A)
B:=B-50	Lock-S(B)
Write(B)	Read(B)
Lock-X(A)	Display(A+B)
Read(A)	Unlock(A)
A:=A+50	Unlock(B)
Write(A)	
Unlock(B)	
Unlock(A)	

P_8: T_1	T_2
Lock-X(B)	
Read(B) (200)	
B:=B-50	
Write(B) (150)	
	Lock-S(A)
	Read(A) (100)
	Unlock(A)
	Lock-S(B) → espera por T_1
	Read(B) (150)
Lock-X(A) → espera por T_2	
...	…

Ambas transacciones se han bloqueado mutuamente (*deadlock*). Cuando ocurre un bloqueo mutuo el sistema debe retroceder una de las dos transacciones, los datos bloqueados se liberan y la otra puede continuar. Lo anterior indica que el bloqueo debe usarse con cuidado. Si se quiere aumentar la concurrencia desbloqueando tan

pronto como se pueda, se puede incurrir en estados inconsistentes, y si no, se puede originar un bloqueo mutuo.

Para controlar este mecanismo de control de concurrencia deben definirse reglas que regulen los bloqueos y desbloqueos, conocidas como protocolo de bloqueo. Los protocolos de bloqueo restringen la cantidad de planificaciones a un subconjunto propio de todas las planificaciones seriabilizables posibles.

Definición:

Sea $\{T_0,\ldots,T_n\}$ un conjunto de transacciones que participan en una planificación S. Se dice que T_i precede a T_j en S, denotado por $T_i \rightarrow T_j$ si existe un dato Q tal que T_i haya tenido un modo de bloqueo A sobre Q, y T_j haya tenido después un modo de bloqueo B sobre Q y COMP(A,B) = false.

Si $T_i \rightarrow T_j$, entonces esto implica que en cualquier planificación en serie equivalente, T_i debe aparecer antes que T_j. Este grafo es similar al de conflictos donde los conflictos entre instrucciones corresponden a modos de bloqueo incompatibles. Se dice que una planificación S es legal bajo un protocolo de bloqueo dado, si S es una planificación posible para un conjunto de transacciones que sigue las reglas del protocolo de bloqueo. Un protocolo de bloqueo garantiza la seriabilidad en conflicto si y sólo si para todas las planificaciones legales la relación asociada es acíclica.

Protocolo de bloqueo en dos fases

Este protocolo garantiza la seriabilidad y se basa en las fases siguientes:
1. Fase de crecimiento: Una transacción obtiene bloqueos.
2. Fase de encogimiento: Una vez que una transacción libere un bloqueo sólo puede hacer liberaciones, no puede adquirir ningún otro.

Un protocolo de bloqueo en dos fases (2PL) garantiza la seriabilidad en conflictos pero no evita el bloqueo mutuo. El protocolo de bloqueo en dos fases se puede refinar para aumentar la concurrencia. Observar las transacciones siguientes:

T_1	T_2
Read(a_1)	Read(a_1)
Read(a_2)	Read(a_2)
…	Display(a_1+a_2)
Read(a_n)	
Write(a_1)	

Si se bloquea de modo X a a_1 desde el inicio en T_1, se elimina la posibilidad de concurrencia con T_2. Se puede primero solicitar un bloqueo S a a_1 y después elevarlo a X, para ello se tienen las sentencias: *upgrade*, que convierte de modo S a X, y *downgrade*, que convierte do modo X a S. Estas sentencias no se pueden usar de manera arbitraria, *upgrade* sólo se puede usar en fase de crecimiento y *downgrade* en fase de encogimiento.

Ejemplo:

T_1	T_2
Lock-S(a_1)	
	Lock-S(a_1)
Lock-S(a_2)	
	Lock-S(a_2)
Lock-S(a_3)	
	display(a_1+a_2)
	unlock(a_1)
	unlock(a_2)
upgrade(a_1)	
…	…

Observar que una transacción que intenta elevar un bloqueo sobre un dato Q puede ser que tenga que esperar.

Cuando una transacción ejecuta read(x) el sistema automáticamente ejecuta Lock-S(x), si una transacción ejecuta write(x) se comprueba si dicha transacción tiene un bloqueo S sobre x en cuyo caso ejecuta upgrade(x) si no se ejecuta Lock-X(x). Existen planificaciones seriabilizables en conflicto que no pueden obtenerse mediante el protocolo de bloqueo en dos fases y se dispone de otros protocolos.

Los SMBDD no sólo manejan bloqueos sino también las responsabilidades de su manejo, los usuarios no necesitan especificar cuándo los datos deben ser bloqueados; los SMBDD tienen este cuidado cada vez que se emite una sentencia de lectura o de escritura.

En sistemas basados en bloqueo, el planificador es un planificador de bloqueo (LM), el manejador de transacciones (TM) envía al LM la información requerida (operaciones, datos, identificador de transacción, etc.) y éste chequea si lo solicitado se puede conceder, en cuyo caso solicita los servicios del procesador de datos y se informa al TM de los resultados. La terminación de una transacción ocasiona la liberación de los bloqueos y el inicio de otra que pudiera estar esperando porque no se les concedieron los bloqueos que solicitó.

Las operaciones de bloqueo y liberación también deben ser coordinadas. El teorema de Eswaran garantiza que cualquier planificación generada por un algoritmo de control de concurrencia 2PL es seriabilizable. Observar que con el protocolo 2PL se permite que otras transacciones puedan bloquear datos que se liberan antes de comprometerse, por lo que si la transacción que liberó sus datos aborta, la otra también debe hacerlo. Debido a esto la mayoría de los manejadores 2PL son 2PL estrictos, o sea, liberan todos sus bloqueos cuando la transacción termina. O sea, el protocolo 2PL libera sus bloqueos en escala y el 2PL estricto los libera juntos cuando termina la ejecución con Commit o Rollback.

Estos protocolos pueden ser extendidos a BD distribuidas, una forma de hacerlo es delegando la responsabilidad del manejo sólo a un sitio, o sea, sólo un sitio tendrá un TMD. Los manejadores de transacciones de los restantes sitios se comunican con éste. Este protocolo distribuido se conoce como protocolo 2PL de sitio primario o 2PL centralizado aunque no es exactamente igual a un protocolo 2PL centralizado propiamente dicho pues tiene que hacer coordinaciones con otros sitios. También existe una extensión de este concepto denominado 2PL de copia primaria que es similar al anterior, pero la copia primaria de unos datos puede radicar en un sitio y la de otros datos en otro sitio. Esta variante reduce la carga de trabajo de un único sitio central y demanda un manejo de directorio más complejo.

El protocolo 2PL distribuido permite manejadores de bloqueo en todos los sitios. Si la BD no está replicada este protocolo coincide con el anterior de copia primaria, y si la BD está replicada se instrumenta el control de réplica ROWA (*Read-Once/Write-All*). Este protocolo se usa en el sistem R*.

ALGORITMOS DE CONTROL DE CONCURRENCIA BASADOS EN HORA DE ENTRADA (TIMESTAMPS)

En los protocolos ya descritos, el orden entre dos transacciones que entran en conflicto está determinado en tiempo de ejecución por el primer bloqueo que solicitan ambas transacciones que implican modos incompatibles. Otro método para determinar este orden de seriabilidad es seleccionar por adelantado una ordenación entre transacciones basada en un esquema de hora de entrada.

A cada transacción T_i del sistema se le asocia una hora de entrada fija única, representada por $TS(T_i)$. El sistema asigna esta hora antes que T_i empiece su ejecución. Si se inicia una nueva transacción T_j entonces $TS(T_i) < TS(T_j)$.

Existen dos métodos sencillos para instrumentar este esquema:

1. Usar el valor de la hora del reloj del sistema.
2. Usar un contador que se incrementa cada vez que se inicia una transacción y considerar como su hora de entrada a este valor.

Las horas de entrada de las transacciones determinan el orden de seriabilidad. Así, si $TS(T_i) < TS(T_j)$ entonces el manejador de transacciones tiene que asegurarse de que la planificación que se obtenga sea equivalente a una planificación en serie en que T_i aparezca antes que T_j.

Para instrumentar esta técnica, se asocian dos valores de hora de entrada a cada dato Q.
1. whe(Q): representa la mayor hora de entrada entre las transacciones que ejecutaría con éxito write(Q).
2. rhe(Q): de manera similar para read.

Como es obvio, estas horas se actualizan cada vez que se ejecutan sentencias de entrada/salida. El protocolo funciona como sigue:

Suponer que T_i solicita read(Q):
1. Si $TS(T_i) <$ whe(Q) quiere decir que T_i necesita leer un valor de Q que ya fue actualizado. Se rechaza la lectura y T_i se restaura (se inicia de nuevo con otra hora de entrada)
2. Si $TS(T_i) >=$ whe(Q), se ejecuta read(Q) y rhe(Q):= max(rhe(Q), $TS(T_i)$)

Suponer que Ti solicita write(Q):
1. Si $TS(T_i) <$ rhe(Q) o $TS(T_i) <$ whe(Q) esto quiere decir que el valor que T_i está produciendo ya fue utilizado. Se rechaza write(Q) y T_i retrocede.
2. Si $TS(T_i) >=$ rhe(Q) and $TS(T_i) >=$ whe(Q) se ejecuta write(Q) y whe(Q):=max(whe(Q), $TS(T_i)$)

Ejemplo:

T_1	T_2
Read(B)	Read(B)
Read(A)	B:=B-50
Display(A+B)	Write(B)
	Read(A)
	A:=A+50
	Write(A)
	Display(A+B)

Valorar si la planificación siguiente es correcta bajo este protocolo:

T_1	T_2
Read(B)	
	Read(B)
	B:=B-50
	Write(B)
Read(A)	
	Read(A)
Display(A+B)	
	A:=A+50
	Write(A)
	Display(A+B)

Este protocolo garantiza seriabilidad en conflictos.

Granularidad

Hasta ahora se ha usado cada dato individual como la unidad en que se realiza la sincronización. Sin embargo, hay circunstancias en las que conviene agrupar varios datos y tratarlos como unidad de sincronización. Por ejemplo, si una transacción necesita acceder a toda la BD, se consume menos tiempo si se bloquea completa.

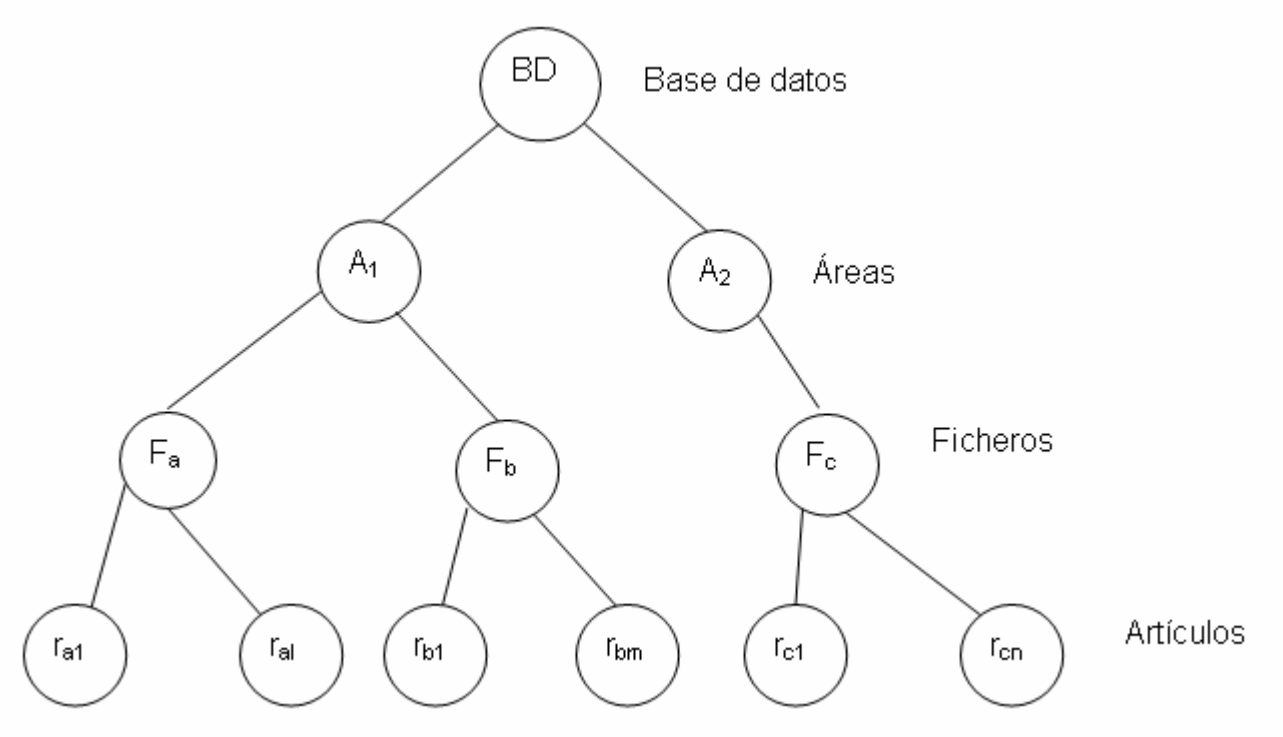
Para permitir varios niveles de granularidad se necesita definir un mecanismo. Esto se logra al establecer una jerarquía de granularidad en la que las granularidades más pequeñas se anidan en las más grandes.

Cada nodo puede bloquearse individualmente con los modos S y X. Cuando se bloquea un nodo con un modo quedan también bloqueados sus descendientes con igual modo. Si por ejemplo la transacción T_i bloquea a F_b, quedan bloqueados $r_{b1}…r_{bk}$. Si otra transacción T_j desea bloquear a r_{b2} debe recorrer desde la raíz hasta determinar si ese bloqueo se concede. Si en el camino hay algún nodo bloqueado con modo incompatible, la transacción T_j que está solicitando el bloqueo debe retroceder.

De manera similar, si otra transacción desea bloquear la BD completa no podrá hacerlo pues T_i tiene un bloqueo sobre parte del árbol. Puede parecer muy restrictivo el uso de la granularidad múltiple, no obstante es usual que se permita junto con un modo de bloqueo que se le llama “de intención”.

Si se desea bloquear un nodo en un modo de intención se hace un bloqueo explícito en los niveles más bajos (más cercanos a la raíz), es decir, en una granularidad más fina. Los bloqueos de intención se colocan en todos los antepasados de un nodo antes que ese nodo se bloquee explícitamente. Así, una transacción no requiere recorrer todo el árbol para determinar si puede bloquear un nodo o no.

Ejemplo:



Existe un modo de intención asociado al modo S y otro al X (IS e IX) respectivamente, también existe el modo SIX (ISX).

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Este protocolo de bloqueo de granularidad múltiple garantiza seriabilidad, cada transacción T_i puede bloquear un nodo Q utilizando las reglas siguientes:

1. Debe respetarse la función de compatibilidad de bloqueo.

2. Primero debe bloquearse la raíz del árbol y esto puede hacerse en cualquier modo.

3. T_i puede bloquear a un nodo Q en el modo S o IS sólo si T_i bloqueó al padre de Q en modo IX o IS.

4. T_i puede bloquear al nodo Q en modo X, SIX o IX sólo si T_i bloqueó al padre de Q en modo IX o SIX.
5. T_i puede bloquear un nodo sólo si no desbloqueó antes ningún otro (o sea, es 2PL).
6. T_i puede desbloquear un nodo Q sólo si ninguno de los hijos de Q está bloqueado por T_i.

Observar que se bloquea de la raíz a las hojas y se desbloquea a la inversa.

Ejemplo: En base a la jerarquía ya mostrada.

T1: Leer r_{a2} de F_a
Bloquear BD en modo IS
Bloquear A_1 en modo IS
Bloquear F_a en modo IS
Bloquear r_{a2} en modo S

T2: Modificar r_{a9} de F_a
Bloquear BD en modo IX
Bloquear A_1 en modo IX
Bloquear F_a en modo IX
Bloquear r_{a9} en modo X

T3: Leer todos los registros de F_a
Bloquear BD en modo IS
Bloquear A_1 en modo IS
Bloquear F_a en modo S

T4: Leer todo la BD
Bloquear BD en modo S

¿Qué niveles de concurrencia se permiten?

T_1, T_3 y T_4 acceden a la BD de manera concurrente.

T_2 puede ejecutarse concurrentemente con T_1, pero no con T_3 ni T_4.

Este protocolo intensifica la concurrencia y reduce la sobrecarga en el manejo de bloqueos. Es útil en aplicaciones que incluyen:

1. Transacciones largas que necesitan bloquear muchos datos, pueden bloquear granos gruesos.
2. Transacciones cortas que necesitan bloquear granos finos.

De esta forma las transacciones largas no gastan tiempo colocando demasiados bloqueos y las transacciones cortas no bloquean posiciones que no vayan a usar.

ESQUEMAS MULTIVERSIÓN

Los esquemas de control de concurrencia tratados hasta ahora garantizan la seriabilidad retardando una operación o abortando la transacción que solicitó la operación. Se pueden tomar otras decisiones si se mantienen copias antiguas de todos los datos.

Lectura y escritura

En los SMBD multiversión cada operación write(Q) crea una nueva versión de Q. Cuando se solicita una operación read(Q) el sistema selecciona una de las versiones de Q para leer. Esta selección debe hacerse de manera que garantice la seriabilidad y es importante que se pueda hacer de forma fácil y rápida. La técnica más usada de los esquemas multiversión es la de hora de entrada. El sistema asocia una hora de entrada estática única a cada transacción T_i, sea esta $TS(T_i)$. A cada dato Q se asocia una secuencia con sus versiones $\{Q_1,\ldots, Qm\}$, cada versión Q_k tiene tres campos:

1. Contenido: el valor de la versión Q_k.
2. whe(Q_k): representa la hora de entrada de la transacción que creó la versión Q_k.
3. rhe(Q_k): representa la mayor hora de entrada de las transacciones que han leído con éxito la versión Q_k.

Con cada sentencia write(Q) se crea una nueva versión de Q, sea la k, de la forma siguiente:

1. Contenido: se asigna el nuevo valor escrito.
2. whe(Q_k):= TS(T_i).
3. rhe(Q_k):= TS(T_i). Este valor se actualiza cuando una transacción T_j lee el contenido de Q_k y rhe(Q_k) < TS(T_j).

El esquema siguiente de hora de entrada multiversión garantiza seriabilidad. Suponer que T_i solicita read(Q) o write(Q). Sea Q_k una versión de Q cuya hora de entrada de escritura es la mayor entre las que tienen su hora menor que TS(T_i)

1. Si la transacción T_i solicita read(Q), entonces el valor devuelto es el del contenido de la versión Q_k.
2. Si la transacción T_i solicita write(Q) y si TS(T_j) < rhe(Q_k) entonces T_i retrocede, de lo contrario se crea una nueva versión de Q.

La regla 1 significa que una transacción lee la versión más reciente, anterior a ella. La regla 2 obliga a abortar una transacción si es demasiado tarde para hacer una escritura. Este esquema tiene la propiedad deseable de que una solicitud de lectura nunca falla y nunca tiene que esperar. Esto tiene mucha importancia práctica pues abundan las aplicaciones en que predominan los reportes de lecturas.

Por otra parte, este esquema tiene dos propiedades indeseables:

1. La lectura de un dato requiere la actualización de su hora de entrada para lectura lo que resulta en dos accesos potenciales a disco.
2. Los conflictos entre transacciones se resuelven con retrocesos en lugar de esperas. Esto puede ser costoso, aunque existe un algoritmo que ofrece una variante de solución a este problema.

Hasta ahora sólo se han tratado las operaciones read y write. A continuación se tratan las operaciones de añadir y eliminar.

1. insert(Q): Inserta Q y le asigna valor inicial.
2. delete(Q): Elimina Q.

Si T_i trata de leer Q después de haber sido eliminado, entonces resulta en un error lógico; igual que si se trata de leer antes de haber sido insertado. También ocurre lo mismo al tratar de eliminar un dato que no existe.

Supresión

¿Cómo afecta la concurrencia la operación de supresión? Sean I_i e I_j instrucciones de T_i y T_j, respectivamente, una palnificación S, y las operaciones I_i e I_j consecutivas en S. Sea I_i = delete(Q), y si I_j es:

I_j = read(Q)			
	I_i e I_j tienen conflictos	Si $I_i \rightarrow I_j$	Error lógico de T_j.
		Si $I_j \rightarrow I_i$	Puede leer con éxito.
I_j = write(Q)			
	I_i e I_j tienen conflictos	Si $I_i \rightarrow I_j$	Error lógico de T_j.
		Si $I_j \rightarrow I_i$	Puede ejecutar write con éxito.
I_j =delete(Q)			
	I_i e I_j tienen conflictos	Si $I_i \rightarrow I_j$	Error lógico de T_j.
		Si $I_j \rightarrow I_i$	Error lógico de T_i.
I_j =insert(Q)			
	I_i e I_j tienen conflictos	Q no existía antes de I_i e I_j, entonces	
		Si $I_i \rightarrow I_j$	Error lógico de T_i.
		Si $I_j \rightarrow I_i$	No hay problemas.
		Q existía antes de I_i e I_j, entonces	
		Si $I_i \rightarrow I_j$	No hay problemas.
		Si $I_j \rightarrow I_i$	Error lógico.

Si se emplea bloqueo en 2 fases se requiere un bloqueo exclusivo a un dato antes de eliminarse. Si se emplea protocolo de ordenación por hora de entrada se debe realizar una prueba similar a la de write.

Inserción

La operación insert(Q) tiene conflictos con delete(Q), read (Q) y write(Q). No se puede hacer read o write antes de que Q exista. Debe ser tratado como write para control de concurrencia.

1. Protocolo de 2 fases: Si T_i realiza insert(Q), a Ti se le da bloqueo exclusivo en el dato Q recién creado.
2. Protocolo de ordenación por hora de entrada: Si T_i realiza insert(Q), los valores rhe(Q) y whe(Q) se ponen a $TS(T_i)$.

El fenómeno fantasma

El fenómeno fantasma o problema *phantom* se presenta cuando no sólo se consideran operaciones de lectura y actualización (BD estáticas) sino también inserciones y eliminaciones (BD dinámicas).

Suponer los esquemas:

CUENTA

C#	Lugar	Balance
339	Vueltas	750
914	Sagua	2308
22	Sagua	1550

A

Lugar	Total
Vueltas	750
Sagua	3858

Suponer transacciones:

T_1: Lee las cuentas de Sagua, suma sus balances y compara con la suma correspondiente en A.

T_2: Inserta una nueva cuenta y actualiza el total en A.

Sea la planificación:

T_1	T_2
Read(914) 2308	
Read(22) 1550	
	INSERT INTO Cuenta VALUES (99, Sagua, 50)
	Read(A.Lugar='Sagua') 3858
	Write(A.Lugar='Sagua') 3908
Read(A.Lugar='Sagua') 3908	
Comp(2308+1550=3858,3908)	
ERROR	

Esta ejecución es conflictiva. El fenómeno fantasma puede ser evitado mediante granularidad ordinaria –de relaciones, no de tuplas– o mediante el protocolo de bloqueo de índices, del cual existen varias técnicas para eliminar este fenómeno bajo los distintos protocolos de control de concurrencia. Cualquier transacción que inserte tuplas debe insertar información en cada índice mantenido de la relación.

MANEJO DE DEADLOCK

Como ya se ha mostrado, los algoritmos de control de concurrencia pueden conducir a un estado de *deadlock*, o sea un conjunto de transacciones esperan unas por otras.

Ejemplo

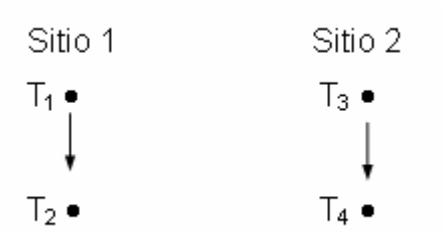
T_1	T_2
Lock-X(x)	
write (x)	
	Lock-X(y)

	write (y)
Lock-X(y) wait	
write (y)	
	Lock-X(x) wait
	write (x)
…	…

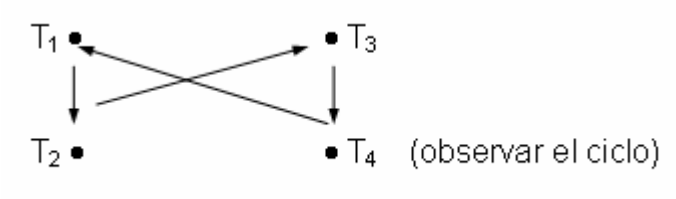
Un *deadlock* es un fenómeno permanente. Si se presenta en un sistema, no se resuelve si no es con una acción externa de parte del usuario, del SMBD o del sistema operativo. Una técnica útil para analizar los *deadlock* son los grafos de espera (WFG). Este es un grafo dirigido que representa la interrelación “espera-por“ entre transacciones. Los nodos de este grafo representan las transacciones concurrentes del sistema, un arco entre T_i y T_j ($T_i \rightarrow T_j$) indica que la transacción T_i está esperando porque T_j libere un dato. En el ejemplo del problema anterior:

Un *deadlock* ocurre cuando este grafo contiene un ciclo. El mantenimiento de este grafo es más difícil en sistemas distribuidos pues dos transacciones que participan en un *deadlock* pueden estar ejecutándose en sitios diferentes de manera que es una *deadlock* global. En sistemas distribuidos no es suficiente que cada SMBDD local forme su grafo de espera local; es también necesario formar un grafo de espera global.

Ejemplo: Considerar cuatro transacciones T_1, T_2, T_3 y T_4 con las interrelaciones de espera siguientes $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. Si T_1 y T_2 se ejecutan en el sitio 1 mientras que T_3 y T_4 en el sitio 2, los grafos de espera locales no muestran el conflicto.



Para detectar el *deadlock* global, hay que incluir las esperas entre sitios.



Hay varios métodos para tratar el *deadlock*. A continuación se muestran algunos métodos que evitan el *deadlock*.

Métodos que evitan el *deadlock*

Los esquemas que evitan el *deadlock* emplean técnicas de control de concurrencia que nunca incurren en *deadlock* o requieren que los planificadores detecten situaciones potenciales de *deadlock* por adelantado y aseguren que no ocurra.

La forma más simple de evitar *deadlock* es ordenar los recursos o unidades de bloqueos y las transacciones siempre adquieren bloqueos en tal orden. Este ordenamiento de unidades de bloqueo puede hacerse globalmente o localmente en cada sitio. Si el sistema es distribuido y el ordenamiento es local, también es necesario incluir en el ordenamiento a los sitios y se requiere que las transacciones que accedan a datos de diferentes sitios adquieran sus bloqueos visitando los sitios en un orden predefinido.

Otra alternativa es hacer uso de las horas de entrada de las transacciones para priorizar transacciones y resolver el deadlock abortando ya sea la de mayor prioridad o la de menor. Para instrumentar este tipo de alternativa, el manejador de bloqueo debe ser modificado como sigue: Si una solicitud de bloqueo de una transacción T_i es

denegada, el manejador de bloqueo no debe forzar a que T_i espere sino que debe aplicarse una prueba. Si T_i pasa con éxito la prueba, puede esperar, de lo contrario una de las dos transacciones involucradas en el conflicto debe abortar. Los métodos más conocidos para estas pruebas son: Esperar-Morir y Herir-Esperar. Ambos métodos se basan en la asignación de horas de entrada a las transacciones y se describen a continuación:

Regla Esperar - Morir: Si T_i solicita un bloqueo sobre un elemento de dato que ya está bloqueado por T_j; T_i espera si y sólo si T_i es más vieja que T_j. Si T_i es más nueva que T_j entonces T_i aborta y se reinicia con la misma hora de entrada.

Regla Herir - Esperar: Si T_i solicita un bloqueo sobre un elemento de dato que ya está bloqueado por T_j, entonces T_i espera si y sólo si T_i es más nueva que T_j sino T_j aborta y concede el bloqueo a T_i.

Estas reglas son definidas en términos de T_i: T_i espera, T_i muere y T_i hiere a T_j. El efecto de herir y morir son los mismos, o sea, abortar. Estas reglas se pueden resumir así:
1. Esperar - Morir: Si TS (T_i) < TS (T_j) entonces T_i espera sino T_i muere.
2. Herir - Esperar: Si TS (T_i)<TS(T_j), entonces T_j se hiere sino T_i espera.
Observar que en ambas reglas la transacción más joven es la que aborta. Estos métodos adicionan un tiempo extra a las ejecuciones de las transacciones.

Detección y solución del deadlock

Es el método de manejo de bloqueo más popular. La detección del *deadlock* se hace sobre el análisis del grafo de espera para determinar si hay ciclos y su solución es: una vez detectado usualmente se hace seleccionando una o más víctimas que deben abortar para romper el ciclo. La selección de una víctima se hace entre las transacciones involucradas en el *deadlock* y no es necesariamente la transacción que originó el ciclo. Una víctima puede ser elegida sobre la base de determinados

criterios como son: la más joven, la que ha adquirido menos bloqueos, la que ha hecho la menor cantidad de cambios, la que aún le falta más para terminar, la que forma parte de más de un ciclo en el grafo de espera, etc.

El proceso de ROLLBACK para la víctima involucra no sólo terminar la transacción y deshacer sus actualizaciones, sino también liberar sus bloqueos y otros recursos del sistema además de los recursos propios de la BD. O sea, el manejador de bloqueo debe prestar servicios no sólo al SMBD sino al sistema completo.

RECOBRADO

Ningún sistema trabaja perfectamente el 100 % de las veces. Los sistemas deben realizar chequeos y controles para reducir la posibilidad de fallos pero también deben considerar un conjunto de procedimientos de recobrado ante fallos que inevitablemente ocurren a pesar de dichos chequeos. Los sistemas de cómputo pueden fallar de muchas formas. Es casi imposible que estén previstas todas ellas. Sin embargo, un buen sistema debe ser capaz de recobrarse automáticamente (sin la intervención del usuario) de los tipos de fallas más comunes.

Muchas fallas ocurren debido a errores de programación y a errores en la entrada de datos pero este tipo de falla se debe tratar con técnicas de Ingeniería de Software o con mecanismos de integridad semántica construidos por los SBD. Ellos están fuera del alcance del rango de problemas que los mecanismos de recobrado resuelven.

Hay tres tipos de fallas que son las más importantes en BD centralizadas, conocidas por: fallas de transacción, fallas del sistema y fallas de los medios.

1. Falla de transacción: Ocurre cuando una transacción aborta.
2. Falla del sistema: Ocurre cuando hay pérdidas o corrupción de los datos en almacenamiento volátil (memoria principal).
3. Falla de los medios: Ocurre cuando parte del almacenamiento estable se ha destruido.

Las técnicas usadas para tratar estos dos tipos de fallas son similares. Se asume que cierta parte del almacenamiento es no confiable (ya sea volátil para fallas del sistema o no volátil para fallas de los medios) y para restaurarlo de la pérdida de datos se deben mantener copias de los datos en diferentes medios de representación.

Para enfrentar este proceso de recobrado, el manejador de datos (DM) se divide en dos componentes: el manejador de cache (CM) que manipula el almacenamiento, y el manejador de recobrado (RM) que procesa read, write, commit, etc. El manejador CM proporciona operaciones *fetch* y *flush* para la transferencia de datos entre almacenamiento estable y volátil. El RM controla las operaciones *flush* para asegurar que el almacenamiento estable siempre tenga los datos necesarios para el recobrado, en caso que se necesite.

La granularidad de los elementos que son escritos en disco es usualmente la página de tamaño fijo (o bloque). Cuando un bloque se escribe en disco hay dos posibles resultados: el valor viejo del bloque se sobreescribe o el nuevo valor se mantiene además del viejo para prever posibles fallas.

Para evitar el acceso a almacenamiento estable para procesar todo read y write, es deseable mantener copias de porciones de la BD en almacenamiento volátil. Para leer un elemento de dato se puede obtener su valor desde la copia en almacenamiento volátil. Para escribir un elemento de dato debe escribirse el nuevo valor tanto en almacenamiento volátil como no volátil. No siempre es posible mantener copia completa de datos en almacenamiento volátil pues esto es costoso y se puede usar también una técnica de bufferización.

Manejador de recobrado

El manejador de recobrado incluye 5 procedimientos:

1. RM - Read (T_i, x): Lee el valor de x para T_i.
2. RM - write (T_i, x, v): Escribe v en x en nombre de T_i.
3. RM - Commit (T_i): Commit de T_i.
4. RM - Rollback (T_i): Aborto de T_i.
5. Restart: Restauración debido a una falla.

Se asume que el planificador invoca operaciones a RM en un orden que produce una ejecución seriabilizable y estricta. Cuando las ejecuciones son estrictas, las escrituras comprometidas se ejecutan en el mismo orden que sus correspondientes transacciones se comprometieron. En particular, el último valor comprometido de x será escrito por la última transacción comprometida que escribió en x.

El planificador produce una ejecución seriabilizable y recobrable. Ejecuciones estrictas evitan abortar en cascada. Para borrar los efectos de una transacción abortada se restaura la BD a las imágenes anteriores a sus escrituras.

Para la descripción posterior del proceso de recobrado se establece la terminología siguiente:
- Imagen anterior de x respecto a T_i: es el valor de x anterior a la escritura de T_i sobre x.
- Imagen posterior de x respecto a T_i: es el valor escrito en x por T_i.

Uso de bitácora

Un RM usualmente almacena información en almacenamiento estable en adición a la BD en sí mismo. Una bitácora es uno de tales tipos de información. Conceptualmente una bitácora es una representación de la historia de ejecuciones, la estructura y contenidos de la bitácora dependen de los algoritmos RM. En general, se puede considerar la bitácora formada por entradas [T_i, x, v] identificando el valor v que la transacción T_i escribió en x.

Las estructuras de datos usadas por la bitácora deben permitir el Restart, determinar para cada x qué entrada de la bitácora contiene el último valor comprometido de x. Una forma de organizar esta información es que la bitácora sea un fichero secuencial y que las entradas en la bitácora sean consistentes con el orden de sus respectivas escrituras. O sea $[T_i, x, u]$ precede en la bitácora a $[T_j, x, v]$ si y sólo si $W_i [x]$ se ejecutó antes que $W_j[x]$.

Como que se asumen ejecuciones estrictas, si $[T_i, x, u]$ precede a $[T_j, x, v]$ en la bitácora y tanto T_i como T_j están comprometidos, entonces T_i se comprometió antes que T_j.

Una bitácora contiene información de operaciones de mayor nivel que simples escrituras. Por ejemplo, una entrada puede describir que “el artículo r fue insertado en el fichero F y los índices fueron actualizados para reflejar esta inserción”. A este tipo de información se le llama bitácora lógica. Con una sola entrada como la anterior se simplifican varias entradas correspondientes a escrituras físicas sobre F y sus índices. Al almacenar operaciones de mayor nivel se almacenan menos entradas y se reduce el tamaño de la bitácora a expensas de adicionar complejidad en la interpretación de sus entradas por el algoritmo Restart. Cuando se usa el término bitácora, se hace referencia a la bitácora física.

Además de la BD y la bitácora, el RM mantiene en almacenamiento estable una o más de las listas activas, comprometidas o abortadas, asociadas a transacciones con esos estados respectivos. Estas listas se almacenan con frecuencia como parte de la bitácora.

Undo y Redo

El RM requiere activar un procedimiento *undo* si permite que transacciones no comprometidas reflejen en la BD valores que las mismas escribieron y sus efectos deben deshacerse para restablecer la BD a su estado consistente.

El RM requiere activar un procedimiento *redo* si una transacción se compromete antes que todos los valores que la misma escribió se hayan almacenado en la BD, en cuyo caso deben repetirse de nuevo para restaurar la BD a su estado consistente. En ambos casos se hace referencia a fallas del sistema, pues estos mismos procedimientos para fallas de los medios tienen otro análisis. Los algoritmos RM se clasifican en 4 categorías:

1. Aquellos que requieren undo y redo.
2. Aquellos que requieran undo pero no redo.
3. Aquellos que requieren redo pero no undo.
4. Aquellos que no requieren ni undo ni redo.

Como que la BD puede contener actualizaciones inapropiadas o puede no contener otras apropiadas, el RM debe contener información suficiente en la bitácora para ejecutar el proceso de restauración en caso que se necesite. Todo RM debe observar las reglas siguientes en su diseño:

Regla Undo: Si la ubicación de x en la BD contiene el último valor comprometido entonces su anterior valor debe ser guardado en almacenamiento estable antes de ser sobreescrito por un valor no comprometido.
Regla Redo: Antes que una transacción pueda comprometerse, el valor que la misma escribió para cada elemento de dato debe estar en almacenamiento estable, ya sea en la BD o en la bitácora.

Las reglas Undo y Redo aseguran que el último valor comprometido de cada elemento de dato esté siempre disponible en almacenamiento estable. La regla Undo asegura que el último valor comprometido de x es guardado en almacenamiento estable –esto es, en la bitácora– antes de ser sobreescrito por un valor no comprometido y la regla Redo asegura que un valor esté en almacenamiento estable en el momento en que se compromete.

Regla de colección de basura: La entrada [T_i, x, v] puede ser removida desde la bitácora si y sólo si: T_i abortó, o si T_i se comprometió pero alguna otra transacción comprometida escribió en x después que T_i lo hizo.

Si T_i se comprometió sólo los últimos valores comprometidos hacen falta para el recobrado. O sea, un valor no se borra de la bitácora aunque se haya guardado en la BD pues si en la misma BD se sobrescribe después, y hay que restaurarlo, es necesario conservarlo.

Algoritmo UNDO/REDO: Es el más complicado de los 4 tipos de RM, y a la vez el más favorable. Un RM que usa undo/redo evita que el CM se vea forzado a hacer escrituras en disco innecesarias, y por tanto minimiza entrada/salida. Por el contrario, un algoritmo no redo generalmente guarda más frecuentemente pues debe asegurar que todos los datos actualizados por una transacción estén en la BD antes que la transacción se comprometa. Un algoritmo no undo no puede guardar en la BD datos no comprometidos. Los algoritmos undo/redo maximizan eficiencia durante ejecución normal a expensas de procesos de recobrado más laboriosos.

Suponer que la transacción T_i escribe el valor v en el elemento de dato x. Con este algoritmo se procede como sigue:
1. RM lee el bloque que almacena x si no está en memoria cache.
2. Almacena v en la bitácora y en memoria cache pero no obliga a CM a guardar este bloque modificado. CM sólo guarda este bloque cuando necesita reemplazarlo por otro. Luego, la acción de guardar los cambios en memoria estable está en poder de CM y no de RM.

Si CM guardó dicho bloque y sucede que T_i abortó o que ocurrió una falla del sistema antes que T_i se comprometa, entonces se requiere un undo. Por otra parte, si T_i se compromete y el sistema falla antes que CM reemplace o guarde el bloque, se requiere un redo.

Se asume que una bitácora física es una lista ordenada de artículos de la forma [T_i, x, v] y que se mantiene de acuerdo a la técnica de colección de basura. Se asume que cada escritura de un dato almacena su valor inicial en la bitácora.

Los procedimientos de RM se describen a continuación:

RM - Write (T_i, x, v):
Paso 1: Adicionar T_i a la lista activa, si no estaba ya.
Paso 2: Si x no está en memoria cache, hacer fetch (x)
Paso 3: Adicionar [T_i, x, v] a la bitácora.
Paso 4: Escribir v en el slot cache ocupado por x.
Paso 5: Informar al planificador de la operación RM-write

RM - Read (T_i, x):
Paso 1: Si x no está en memoria cache, hacer fetch (x)
Paso 2: Retornar el valor de x en el slot cache del planificador.

RM - Commit (T_i):
Paso 1: Adicionar T_i a la lista commit.
Paso 2: Informar al planificador que T_i se comprometió.
Paso 3: Eliminar T_i de la lista activa.

RM - Abort (T_i):
Paso 1: Para cada elemento de dato x actualizado por T_i:
Si x no está en memoria cache, asignarle un slot.
Copiar la imagen anterior de x por T_i en el slot
Paso 2: Adicionar T_i a la lista abort (o rollback)
Paso 3: Informar el planificador que T_i abortó.
Paso 4: Eliminar T_i de la lista activa.

Restart:

Paso 1: Liberar la memoria cache.

Paso 2: Sea redone: ={}, undone:= {}

Paso 3: Comenzar con la última entrada en la bitácora y explorar hacia atrás.

Para cada entrada [T_i, x, v],

Si x $\notin$ (redone $\cup$ undone), entonces

Si x no está en la memoria cache, asignarle un slot.

Si T_i está en la lista commit,

copiar v en el slot cache y hacer redone:=redone $\cup$ {x}

En caso contrario,

Si T_i está en la lista de rollback o en la lista activa pero no está comprometida, copiar la imagen anterior de x por T_i en el slot cache y colocar undone:= undone U {x}.

Paso 4: Para cada T_i en la lista commit, si T_i está en la lista activa, removerla.

Paso 5: Informar el planificador que Restart concluyó.

Puntos de chequeo (*checkpoint*)

Los puntos de chequeo son actividades que escriben información en almacenamiento estable durante una operación normal para reducir el monto de trabajo de Restart. Cuando ocurre un fallo del sistema es necesario consultar la bitácora para determinar cuáles son las transacciones que necesitan volver a hacerse y cuáles son las que necesitan deshacerse. Es necesario revisar toda la bitácora para determinar esto. Este enfoque tiene dos problemas:

1. La búsqueda consume tiempo.
2. La mayor parte de las transacciones que necesitan volver a hacerse ya escribieron sus actualizaciones en la base de datos. Aunque volver a hacerlas no causa ningún daño, hace que la recuperación tarde más.

Para reducir estos tipos de gastos extra, se introduce el concepto de puntos de chequeo o verificación. Para esto se realizan las acciones siguientes:

1. Grabar en memoria estable todos los registros de bitácora que actualmente residen en memoria principal.
2. Grabar en disco todos los bloques modificados de los registros intermedios (buffer).
3. Grabar un registro de bitácora <*checkpoint*> en memoria estable.

La presencia de un registro <*checkpoint*> en la bitácora permite al sistema definir su procedimiento de recuperación. Cualquier modificación de la base de datos que haga T_i debe haberse escrito en la base de datos, o antes del punto de verificación, o como parte del punto de verificación. Por tanto, en el momento de la recuperación no es necesario realizar una operación redo en T_i. Así, después que haya ocurrido un fallo, el sistema de recuperación examina la bitácora para determinar cuál fue la última transacción T_i que comenzó a ejecutarse antes del último punto de verificación.
<T_j commit>
<checkpoint>
<T_i start>
Una vez identificada T_i, sólo será necesario aplicar las operaciones redo y undo a la transacción T_i y a todas las T_k que comenzaron a ejecutarse después de T_i. El resto de la bitácora puede ignorarse. Las operaciones de recuperación que se van a realizar dependen de si se va a utilizar la técnica de modificación inmediata o diferida.

Inmediata:

1. Ejecutar redo(T_l) para todo T_l que pertenece a T ($T=T_i \cup T_k$) cuyo registro <T_l commit> correspondiente aparezca en la bitácora.

2. Ejecutar undo(T_l) para todo T_l que pertenece a T ($T=T_i \cup T_k$) cuyo registro <T_l commit> no aparezca en la bitácora.

Diferida: No se aplica undo.

Existen otras técnicas de recuperación como es el uso de doble paginación basada en los conceptos de paginación de sistemas operativos. Las técnicas de doble paginación hacen las recuperaciones ante caídas más rápidas, aunque fragmentan los datos y necesitan hacer recolección de basura con bastante frecuencia. Además de ser un proceso complejo, también es muy difícil de adaptar para ejecutar de forma concurrente varias transacciones. En la bibliografía recomendada se pueden encontrar detalles del mismo. Es posible utilizar los esquemas basados en bitácora con pocas modificaciones en un sistema de bases de datos concurrente sin utilizar la doble paginación.

Exploración de bitácora:

Al introducir los puntos de verificación no se tuvo en cuenta la concurrencia, por lo que solamente se consideraron las transacciones siguientes durante la recuperación:
1. Transacciones que comenzaron después del último punto de verificación.
2. Transacción que estaba activa (si hubo) en el momento del último punto de verificación.

Si las transacciones pueden ejecutarse concurrentemente, la situación es más compleja. Si el punto de verificación deber ser <checkpoint L> donde L es la lista de transacciones activas en el momento del punto de verificación. Cuando el sistema se recupera de una caída construye dos listas: la lista de deshacer (undo), y la lista de volver a hacer (redo). Se construyen de la manera siguiente:
1. Inicialmente están vacías
2. Se examina la bitácora hacia atrás, registro por registro, hasta que se encuentre el primer registro <checkpoint>.
2.1. Por cada registro que se encuentre en la forma <T_i commit> se añade T_i a la lista de volver a hacer.
2.2. Por cada registro que se encuentre de la forma <T_i start> si T_i no está en la lista de volver a hacer, se añade T_i a la lista de deshacer.

3. Cuando se han examinado todos los registros de bitácora apropiados, se comprueba la lista L. Para cada transacción T_i en L, si T_i no está en la lista de valores a hacer, entonces se añade T_i a la lista de deshacer.

Una vez construidas las listas de volver a hacer y la de deshacer, la recuperación procede así:
1. Se vuelve a examinar hacia atrás la bitácora del registro más reciente y se realiza undo(T_i) para cada T_i en la lista de deshacer.
2. Se continúa la exploración de la bitácora hacia atrás hasta que se haya localizado el registro <T_i start> para todas las T_i en la lista de volver a hacer.
3. Se examina la bitácora hacia adelante y se realiza redo(T_i) por cada T_i en la lista de volver a hacer.

GESTIÓN DE TRANSACCIONES EN SBD MÚLTIPLES

El manejador de transacciones en Sistemas de Bases de Datos Múltiples (SBDM) mantiene dos tipos de transacciones: locales, que son ejecutadas por cada SMBD, fuera del control del SBDM, y globales, que están bajo el control del SBDM. Un SMBD no puede comunicarse con otro directamente para sincronizar la ejecución de una transacción global activa en varias localidades.

El criterio de corrección para la ejecución concurrente es la seriabilidad. Puesto que el SBDM no tiene control sobre la ejecución de las transacciones locales, cada SMBD debe usar algún esquema de control de concurrencia para asegurar que su planificación es seriabilizable, ya sea bloqueo en 2 fases o de horas de entrada. Además, el SMBD local debe protegerse de bloqueos locales. La garantía de seriabilidad local no es suficiente para asegurar la seriabilidad global.

Ejemplo: T_1 y T_2 acceden y actualizan datos A y B residentes en L_1 y L_2. Suponer planificaciones locales seriabilizables. Es posible en L_1 que T_2 siga a T_1 y L_2 que T_1 siga a T_2. Así, el resultado es una planificación no seriabilizable global.

Una manera de impedir esta dificultad es prohibir transacciones globales en la actualización de los datos. Las transacciones globales sólo hacen lecturas locales y escriben. Esta restricción limita la capacidad de acceso del SBDM. La situación llega a ser más complicada cuando a las transacciones globales se les permite leer y escribir datos locales.

Una condición suficiente para la seriabilidad global será que en todas las planificaciones locales el orden de seriabilidad de la transacción global sea la misma. Esto es, sean T_i y T_j dos transacciones globales que se ejecutan en la localidad L_k. Suponiendo que la planificación local en L_k es equivalente a una planificación en serie en la cual T_i aparece antes que T_j. Entonces, lo mismo se debe mantener para todas las demás localidades donde se ejecutan T_i y T_j.

Una manera de asegurar las mismas planificaciones en serie es requerir que el SMBD local provea al SBDM con su información de planificación local. El SBDM puede usar esta información para garantizar la seriabilidad global. Sin embargo, esto requiere cambios en el diseño del SMBD local resultando en una violación de la autonomía local.

Existen varios esquemas para asegurar la seriabilidad global sin sacrificar la autonomía local. La corrección está asegurada si cada SMBD local utiliza el estricto protocolo de bloqueo de 2 fases, y los bloqueos de cada transacción global se liberan sólo cuando la transacción ha terminado su ejecución. Si el SMBD local utiliza un tipo diferente de esquema de gestión de transacción, entonces el SBDM debe usar alguna forma de esquema de control de concurrencia para asegurar la seriabilidad global.

Un esquema simple sería el de grafo de transacción. Un grafo de transacción es un grafo bipartito indirecto con un grupo de nodos que constan de nombres de transacciones globales y nombres de localidades (locales). Si T_i accede a algunos

datos en L_k se formará en el grafo una arista indirecta entre T_i global y L_k. La seriabilidad global asegura cuando el grafo de transacción no contenga ciclos indirectos.

CAPÍTULO 4. OPTIMIZACIÓN DE CONSULTAS

La optimización de consultas es tanto un desafío como una oportunidad para los sistemas relacionales. Es un desafío puesto que es imprescindible en un SMBD Relacional, y una oportunidad porque es una de las “fortalezas” de la aproximación relacional ya que las expresiones son de alto nivel semántico y permiten la optimización.

Por otra parte, puesto que el usuario no necesita realizar la optimización de sus procedimientos, sino que es el propio sistema el que realiza dicho proceso automáticamente, es necesario tener en cuenta todas (o casi todas) las posibilidades de optimización de una consulta dada. Se dice que los lenguajes de consulta relacionales son no procedimentales en el sentido de que no se suministra un procedimiento concreto para su resolución sino que sólo se especifica el qué se quiere obtener.

Un buen optimizador siempre dispondrá de más y mejor información que un programador que tuviera que realizar la tarea manualmente, específicamente información estadística tal como cardinalidad de los dominios y de las relaciones base, el número de valores distintos de cada atributo, el número de ocurrencias de un determinado valor en un atributo, etc. Un sistema automático será capaz de asegurar la estrategia más eficiente de implementación de una determinada consulta. Si, estadísticamente hablando, los valores cambian con el tiempo se hará necesaria una reoptimización, un reproceso de la optimización previa, tema éste que en sistemas no relacionales obligaría a la reprogramación de las rutinas establecidas para el caso.

El optimizador es un programa, al fin y al cabo, y es más paciente que el más paciente de los programadores, y capaz de considerar literalmente cientos de diferentes estrategias de implementación, mientras que un humano consideraría, a lo sumo, tres o cuatro. Al ser accesible por todos los usuarios de forma transparente

éstos pueden obtener las ventajas del programador más experimentado y habilidoso sin necesidad de requerir sus servicios explícitamente. El optimizador es una de las características que más “fuerza” proporciona al modelo relacional.

La optimización de consultas incluye cuatro pasos:
1. Transformación de la consulta en alguna forma interna:
2. Conversión a Forma Canónica
3. Elección de procedimientos de bajo nivel
4. Generación de Planes de Consulta y elección del más económico

PREMISAS BÁSICAS

Sean:
- R una relación.
- T_R el número de tuplas de R.
- B_R el número de bloques en los cuales se acomodan todas las tuplas de R. Además B_R es mucho menor que T_R.

Así, el número de tuplas por bloque se puede obtener por la expresión T_R / B_R. Se asumen bloques de 1024 bytes. Por ejemplo, si R tiene un millón de tuplas de 100 bytes de longitud, T_R = 1.000.000; luego B_R = 100.000.

El costo de leer o escribir una relación es el número de bloques que deben ser leídos desde disco a memoria principal o escritos en sentido inverso. Para las operaciones de lectura y/o escritura masiva, si el almacenamiento es muy compacto, significa un costo bajo; en otro caso implica alto costo.

En cuanto al uso de índices, las estructuras de hash o B-tree son las ideales, porque permiten ejecutar la selección de tuplas rápidamente. Se asume que toda relación R tiene un índice en un atributo A. Luego $\sigma_{A=C}$ (R) tiene aproximadamente un costo del número de bloques donde existen tuplas de R que cumplen tal condicion.

Considerando descartable el número de bloques leídos para obtener el índice, pues dicho número es usualmente menor o igual que el número de bloques de R que se deben acceder.

Si se tienen índices cluster en un atributo A, el número de bloques guardando tuplas con A = *c*, es aproximadamente el número de bloques en el cual esas tuplas pueden ser almacenadas. En el mejor de los casos, un solo bloque para todos, en el peor un bloque por tupla.

Si se tienen índices no cluster, se puede asumir que las tuplas de R que cumplen la condición A=*c*, aparece cada una en bloques separados. Luego, los bloques leídos equivalen al número de tuplas que cumplen tal condición. Por ejemplo, en un índice B-tree denso solo será coincidencia si aparecen en el mismo bloque físico dos tuplas con el mismo valor para A.

El tamaño de la imagen, denotado por $I_{R.A,}$ es el número esperado de diferentes valores de A hallados en R. Asumiendo que los valores son igualmente probables de ocurrir, se puede estimar el número de bloques recuperados en respuesta a $\sigma_{A=C}$ (R):

- Si tenemos un índice no-cluster, T_R / I_R es el número esperado de tuplas recuperadas.
- Si hay un índice cluster en A, se recuperan cerca de B_R / I_R bloques, pues las tuplas escogidas están ocupando ese número de bloques.

Ejemplo:

Sea R con T_R = 1.000.000, B_R = 100.000, $I_{A\,R}$ = 50000
Sin cluster → 1.000.000 tuplas / 50.000 diferentes valores de A. = 4 lecturas.
Con cluster → 100.000 bloques / 50.000 valores de A = 2 lecturas.

BASES DE LA OPTIMIZACIÓN

Hay dos tipos básicos: la manipulación algebraica y la estimación de costos. La primera es independiente de los datos y de su estructura física actual, y es tratada en este capítulo después de la segunda, la cual se mide por los accesos a disco, y considera, por ejemplo, el uso de índices. Esta última selecciona entre varias alternativas, la mejor estrategia para los datos y las estructuras de almacenamiento que se usen.

OPTIMIZACIÓN POR ESTIMACIÓN DE COSTOS

Sea la relación AB con atributos A y B. Sea la relación BC con atributos B y C. Ahora observe un pequeño ejemplo de manipulación algebraica, tema que es tratado más adelante en este capítulo.

Sean:

F_1. $\sigma_{A=d}$ (AB $\bowtie$ BC)

F_2. ($\sigma_{A=d}$ (AB)) $\bowtie$ BC

La consulta F_1 es más veloz si primero se ejecuta la selección y despues el join, independiente de la presencia o ausencia de índices en sus atributos. En general, los join son más caros que las selecciones, porque requieren acceder (al menos una vez) a todo bloque en los que residan las tuplas de estas dos relaciones. Además, se requiere almacenar resultados temporales. Por otro lado, la relación (AB $\bowtie$ BC) puede ser mucho mayor que la relación AB o que la relación BC, si la tupla típica de AB tiene un valor B que es compartido por varias tuplas de BC. En promedio, tomar primero la selección reduce el número de tuplas en el primer argumento del join por un factor I_A, lo que provoca bajar, una cifra significativa, por este factor el tamaño del resultado del join.

La selección en la fórmula F_1 es de menor costo que en el caso F_2 sólo si el resultado del join es mucho más pequeño que la relación AB; o sea, si la tupla típica de AB

tiene una probabilidad baja de hacer join con una tupla de BC, lo cual es un caso poco común. En la mayoría de los casos, el costo de la selección en la opción F_2 no es mayor que en la fórmula F_1; inclusive puede ser mucho menor si el resultado del join es mucho más grande que AB y/o si hay un índice para A en la relación AB que pueda ser usado en el caso F_2 pero no en el caso F_1.

Una buena práctica es hacer la selección tan pronto como sea posible. Esta heurística tiene sus excepciones. Si BC está vacía, el producto cartesiano será vacío y se hace innecesaria la selección.

Optimización de la selección en SYSTEM R:

Sea la consulta QSR:

SELECT $A_{1,...}, A_n$
FROM R
WHERE C_1 AND C_2 AND …;

Donde:
C_i son las condiciones que involucran R y que pueden a su vez estar conformadas por subcondiciones conectadas por AND, OR, NOT, IN, etc.; pero con una estructura diferente a C_1 AND C_2.

Este algoritmo exige que las condiciones se descompongan tan elementalmente como sea posible en condiciones conectadas por AND. SYSTEM R toma ventaja de los índices siempre que sea posible. Por ejemplo, si existe C_i de la forma A = *c* y hay un índice en A, se obtienen las tuplas que cumplen esta condición, luego examina en las tuplas obtenidas las demás condiciones.

Ejemplo:

CLIENTE (nombre, dirección, balance)
ORDEN (# Orden, fecha, cliente)

ITEM (# orden, producto, cantidad)

Alternativa de diseño físico: Usar índice cluster entre ORDEN e ITEM por #orden, e índices no cluster en nombre, cliente y producto.
Se asume que cada cliente ha colocado varias órdenes y que cada orden incluye varios items.

Algoritmo de optimización para consultas simples

Es un sistema enumerativo; es decir, existe un número de opciones en un menú preseleccionado; el procesador de consultas estima el costo de cada opción y escoje la mejor.
En situaciones complejas, por ejemplo una consulta con varias reuniones naturales, pueden existir miles de opciones. En un caso simple el número de opciones es poco.
El diseñador del optimizador parte del presupuesto de que el tiempo consumido en optimizar una consulta típica será compensado en el tiempo de ejecución de esta; lo que es altamente probable en relaciones de gran tamaño.
Como no es posible enumerar todos los modos imaginables de implementar una consulta, system R, como cualquier sistema de optimización, limita el espacio de estrategias consideradas incluyendo únicamente aquellas de la forma: seleccionar una de la condiciones C_i, hallar todas las tuplas que satisfacen la condición y luego examinar estas tuplas resultantes para evaluar cuáles de ellas satisfacen las otras condiciones. system R también considera estrategias en las que se comienza examinando todas las tuplas de R y se analiza cuáles de ellas satisfacen todas las condiciones.
Algoritmo
Entrada:
Una consulta de la forma QSR.
Información acerca de qué índices existen en la relación R.
T_R: número estimado de tuplas.
B_R: número mínimo de bloques requeridos para almacenar R.

I_R: tamaño estimado de la imagen para los índices.

Salida:

Una manera de computar la respuesta a la consulta.

Método

Se considera la lista de alternativas para obtener R completa o, aplicando una condición, obtener una selección de R.

Después de seleccionar un método, se aplican las condiciones restantes a las tuplas obtenidas.

Para los métodos que involucran la selección de cuál índice o condición usar se deben considerar todas las posibles alternativas. El costo se juzga en base a T, B e I. La salida del algoritmo es el método de más bajo costo. Las opciones se listan en orden de costo, empezando por las de costo más bajo. System R asume que sólo hay un índice entre las columnas C_i. Para una consulta con condiciones que involucren varios índices (por ejemplo: Where A = 1 and B = 17, con índices en A y B), es mejor intersectar en memoria la colección de apuntadores (índices) obtenidos por A = 1 y por B =17; luego acceder el disco sólo para aquellos apuntadores que pertenezcan a la intersección.

Opción 1: obtener tuplas de R que satisfacen una condición de la forma A =*c*. Donde A tiene un índice cluster. Sea I_R el tamaño de la imagen del índice; luego, el número de bloques que deben leerse para obtener estas tuplas será igual al número de bloques que ellas ocupen si se acomodan de una manera compacta. Asi el número de bloques leídos para este método será B/I.

Opción 2: usar un índice cluster en un atributo A, donde A θ C es una de las condiciones C_i y $\theta \in$ (<, <=, >, >=). A continuación se aplican las condiciones restantes al resultado. En este caso se requiere acceder cerca de B_R / 2 bloques, pues en promedio se requiere leer cerca de la mitad de las tuplas, y con un índice cluster estas se almacenan compactas en cerca de B/2 bloques.

Nota: el caso donde θ equivale al operador $\neq$ es omitido aquí, pues este símbolo implica selección de todas las tuplas y se está suponiendo una selección más limitada.

Opción 3: si hay un índice no cluster que coincida con una condición A =*c*, se usa ese índice para hallar todas las tuplas que cumplen tal condición y se aplica las otras condiciones a las tuplas resultantes. Siendo I_R la imagen del índice, deben recuperarse T_R / I_R tuplas, probablemente desde bloques diferentes.

Opción 4: si R está almacenado en un archivo independiente, se puede simplemente leer todas las tuplas y aplicar las C_i a estas. El costo será B_R número de bloques donde está esparcida R.

Opción 5: si R no está almacenada independientemente, pero tiene un índice cluster en un atributo o en una colección de atributos, sin importar que no estén involucrados en la condición de la consulta, se usa tal índice para obtener todas las tuplas de R y luego aplicar las condiciones a ellas. Su costo también es B_R, pues cualquier índice cluster garantiza que se pueden obtener todas las tuplas en no más lecturas que bloques utilizados en almacenarlas.

Opción 6: si hay un índice no cluster en un atributo A y A θ C es una condición donde $\theta \in$ (<, >, >=, <=); se usa el índice para obtener las tuplas de R que satisfagan la condición y se aplica las otras condiciones al resultado. El costo es $T_R / 2$, pues se espera recuperar cerca de la mitad de los registros de R. Además las tuplas estarán dispersas independientemente en los bloques.

Opción 7: Use un índice no cluster de cualquier clase para hallar las tuplas de R y aplicar todas las condiciones a ellas. Su costo es T_R.

Opción 8: si ninguna de las anteriores opciones es posible, se deben recuperar entonces todos los bloques que podrían contener tuplas de R. El costo de este método es T_R o quizás más, si no se puede asegurar cuáles bloques contienen tuplas de R.

Ejemplo:

```
SELECT #orden
FROM Item
WHERE cantidad >= 5 AND producto = ‘Brie’;
```

Supuestos: Índice cluster para ITEM por # orden e índices no cluster en producto y cantidad.

Suponga 1 000 tuplas en ITEM, T_R =1 000 y que se acomodan 10 tuplas por bloque, B_R =100. Sea la imagen de producto I_R = 50; Es decir, hay 50 diferentes tipos de productos en las órdenes. Para este ejemplo son irrelevantes las imagenes de los otros índices.

Opción 1: no factible; no hay índices cluster en las condiciones de nuestra consulta.

Solución

Opción 2: idem a la opción 1.

Opción 3: se tiene el índice no cluster sobre producto que hace parte de las C_i. El costo $T_R / I_R = 1\ 000/50 = 20$.

Opción 4: Leer todas las tuplas. Si los items están almacenados independientemente B_R =100. Si los items tienen un cluster con órdenes, no es aplicable la opción 4.

Opción 5: Si hay cluster entre ORDEN e ITEM. $B_R = 100$.

Opción 6: usar el índice cantidad. Costo aproximado $T_R / 2 = 500$.

Opciones 7 y 8: Costo $T_R = 1\ 000$, pero no son aplicables en el ejemplo.

El menor costo lo da la opción 3, donde se obtienen aproximadamente 20 tuplas que cumplen las C_i; de ellas se examinan cuáles cumplen con la condición de que la cantidad sea por lo menos cinco, además que se encuentran en bloques diferentes.

Calendario el producto cartesiano

Hay una variedad de estrategias disponibles y el optimizador requiere estimar la de menor costo para una situación dada. Generalmente, el producto y la reunión natural son costosos comparados con operaciones sobre una sola relación, tal como la selección y la proyección. Luego, siempre que sea posible, debe minimizarse su costo. Se introducen dos nuevos parámetros: U y M.

Sea U el número de bloques necesarios para almacenar el resultado de lo calculado. Si el resultado debe ser escrito a disco, U bloques accesados serán necesarios para almacenar la salida. Este valor domina frecuentemente al costo total en productos y reuniones.

Sea M el número de bloques que pueden acomodarse en memoria principal a la vez. Nótese que un valor pequeño para M obliga a leer varias veces el mismo bloque, incrementando el costo de la operación más allá de lo necesario para leer los argumentos y escribir los resultados.

Suponga que se necesita calcular RxS. Se asumen R y S almacenadas independientemente pero compactas, tal que para ser leídas se necesitan B_R y B_S accesos. Si este no es el caso, es usualmente más eficiente leer las tuplas de la relación que no están compactas y hacer copias de ellas, lo que implica un costo adicional de T_R accesos para leer a R, y de B_R accesos para escribir una copia. Se procede de una manera similar en el caso de S.
El algoritmo consiste en un ciclo doble:

FOR cada tupla μ en R DO
FOR cada tupla ν en S DO
sale la tupla $\mu\nu$

Podría, por supuesto, invertirse el rol de los ciclos.

Si una relación cabe en memoria principal,

asumir que $B_S < B_R$. El caso contrario se maneja simétricamente. Si $M > B_S$; entonces S se ajusta en memoria principal. Se puede leer toda S con B_S accesos y luego leer cada bloque de R y descartarlo cuando se lea el siguiente. Para cada bloque de R se ejecuta el ciclo externo, sin costos adicionales de acceso para el ciclo interno pues todas las tuplas están en memoria. Así, el costo para leer la entrada es de $B_S + B_R$. Y el costo del producto es igual a la suma del costo de la entrada más U bloques para la salida.

Se puede estimar U en términos de R y S:
Sean:
L_R: número de bytes para cada tupla de R.
L_S: número de bytes para cada tupla de S.

B: bytes por bloque.

El resultado R x S tendrá tuplas que son la concatenación de tuplas de R con tuplas de S.

$L_{R\,X\,S} \cong L_R + L_S$. bytes.

Resultan $T_R \cdot T_S$ tuplas de salida; luego el número de bloques necesarios para U:

$$U = \frac{T_R \,.\, T_S \;\; (L_R \;\; + L_S \;\;)}{b}$$

U es aproximado, pues cada bloque de tamaño b requiere espacio para información de control y para área libre que, entre otros factores, minimice el encadenamiento.

Note que $B_R \cong (T_R \,.\, L_R) / b$ y $Bs \cong (T_S \,.\, L_S) / b$ luego.

$U = (T_R \,.\, T_S\, L_R.) / b + (T_R \,.\, T_S\, L_S) / b \Rightarrow$

Costo de salida: $U = B_R \,.\, T_S + T_R.\, B_S.$

El costo total será: $B_R (T_S + 1) + B_S (T_R. + 1)$

Ejemplo: Suponga R = 5000 tuplas acomodadas en 500 bloques y S = 1000 tuplas acomodadas en 100 bloques. Sea M = 101.

Se puede leer toda S en memoria y dejar un bloque disponible para R en memoria.

Costo: 500 x 1 001 + 100 x 5 001 = 1.000.600

Lo que tiene sentido, pues resultan cerca de 5 millones de tuplas en la respuesta, cada tupla resultante es la concatenación de una tupla de R con una tupla de S e individualmente ocupan una décima parte de un bloque, entonces se pueden acomodar cinco de las resultantes por bloque.

Productos de relaciones que no caben en menoria principal (relaciones grandes)

Se hace necesario mover varias veces dentro y fuera de memoria principal a algunos de los bloques. Se asume $B_S < B_R$, pero $M < B_S$. Un desempeño casi óptimo se

logra dividiendo S en segmentos de M-1 boques cada uno. Se lee un segmento de S y en el bloque restante de memoria principal se leen uno por uno los bloques de R. Así se obtiene el producto de cada tupla en el segmento de S en turno con cada tupla de R. Se repite este proceso para cada segmento, hasta lograr el producto completo.

FOR cada segmento *s* de la relación S DO
FOR cada tupla ν en segmento S DO
FOR cada bloque B de la relación R DO
FOR cada tupla μ del bloque B DO
sale la tupla $\mu\nu$

Se lee cada bloque de S una vez y cada bloque de R una vez por segmento. Si el número de segmentos es aproximadamente: $\frac{B_S}{M-1}$

Costo de entrada: $B_R(\frac{B_S}{M-1}) + B_S$

Costo de salida: $B_R\, T_S + T_R\, B_S$

Costo Total: $B_R(T_S + \frac{B_S}{M-1}) + B_S(T_R + 1)$

Note que esta expresión de costo total es una generalización de la de costo total cuando la relación cabe en memoria principal. Reemplazando Ts + 1 por $T_S + \frac{Bs}{M-1}$; es decir, Ts + el número de segmentos de S.

Ejemplo: el mismo caso del ejemplo anterior, pero M =11; R =5 000 tuplas S =1 000 tuplas. B_R= 500 bloques. B_S =100 bloques.
Dividiendo S entre Bs / (M -1) segmentos = 100 / 10 = 10 segmentos.
Costo: 500 x (1000 + 100/10) + 100 x 5001 = 1.005.100.
Comparado con el caso anterior, U continúa siendo el factor de más peso aunque el costo de entrada se ha incrementado. Esto es más significativo en los join, pues en ellos la salida es generalmente más pequeña que para el producto, pero si no se

tiene cuidado, el costo de entrada puede llegar a ser tan grande como para el producto.

Un límite más bajo para costos de entrada

Evidentemente, el costo de salida no puede ser mejorado, porque se asume que toda tupla de la salida debe ser escrita. ¿Cómo saber cuál es el costo de entrada más bajo? En realidad, es posible mejorar el costo de una forma no significativa. El teorema que sigue se aplica a cualquier operación, ya sea una reunión natural, como si es un producto; esto es, si cada tupla de una relación debe hallar cada tupla de la otra relación en memoria principal.

TEOREMA: si se ejecuta un producto cartesiano de las relaciones R y S en una memoria que puede guardar M bloques, entonces el número de bloques requeridos para lograr leer R y S es al menos: $\frac{B_R B_S}{(M-1)}$

Prueba

Para obtener una salida $\mu\nu$ es necesario que estén simultáneamente en memoria principal una tupla μ de R y una tupla ν de S. Para cualquier bloque β de R en memoria principal, el beneficio de este evento para obtener tuplas resultantes no puede ser mayor que el número de tuplas que contenga dicho bloque, multiplicado por el número de tuplas de S que en ese momento estén en memoria principal. Este número no será mayor que (M -1) (Ts/ Bs) (Tr / Br). Idem si se lee un bloque de S.

Para calcular el producto completo se debe obtener un número de tuplas tan grande como las que se requieren para la salida, o sea, TrTs. Sea A el número de bloques que se requiere usar como entrada mientras se calcula RXS.

Entonces, $T_S T_R \leq A(M-1)\frac{T_S}{B_S}\frac{T_R}{B_R}$, pues es posible que sea necesario leer varias veces el mismo bloque. Dado lo anterior se puede obtener: $A \geq \frac{B_S B_R}{M-1}$ Número de bloques usados como entrada.

Estimando el costo de salida para los join

Ahora se consideran los algoritmos para calcular la reunión natural de dos relaciones. Si interesan los equijoin, se usa la misma técnica renombrando al atributo común en una de las relaciones, convirtiéndola en una reunión natural.
Si se desea un θ join, donde θ no es la igualdad; por lo general, no es posible superar el algoritmo para el producto de grandes relaciones que no caben en memoria; aunque se ahorra en el tamaño de la salida si muchas tuplas del producto no cumplen la condición del join.

Un modelo estadístico para las relaciones:

Sean AB (a, b) y BC (b, c) dos relaciones. Un factor crítico es cuán grande será el resultado. Se vio anteriormente que la decisión de ejecutar una selección antes que un join depende del tamaño del resultado del join. Este tamaño, a su vez, depende de cuantas tuplas de BC hacen join con una tupla dada ab de la relación AB. No existe una respuesta general para esta pregunta. Se propone un modelo razonable para una relación “típica” y se analiza qué dice dicho modelo.

En primer lugar, se asume que cada atributo de la relación tiene un dominio finito asociado, el conjunto de valores “probables” de aparecer allí. Este dominio puede ser tomado del conjunto de elementos “activos” del dominio potencialmente infinito. Por ejemplo, la relación CLIENTE en el atributo nombre de la BD. YVCB, en el momento de la reunión natural, tiene como dominio el conjunto de clientes de YVCB. Se asume que el conjunto de tuplas potenciales de una relación dada R es el producto cartesiano de los dominios de todos sus atributos. Si R tiene T_R tuplas, y este producto tiene n tuplas, se asume que el valor común de R es un conjunto T_R aleatorio escogido dentro de las n posibles tuplas. Así, cada tupla potencial tiene una probabilidad T_R / n de ser escogida.

Finalmente, se asume que cuando se toma la reunión natural de dos relaciones R y S, los atributos del mismo nombre “significan” lo mismo y por ello se seleccionan sus valores del mismo dominio. Decisión muy sutil, especialmente en el caso de la reunión natural que se origina de un equijoin renombrado, cuyos atributos de igualdad realmente no tengan nada que ver el uno con el otro. Por ejemplo: Hallar dos clientes de YVCB cuyo balance iguala la cantidad de algún item ordenado por algún cliente.
Sin embargo, si el o los atributos del join se refieren a la misma noción, esta premisa tiene una alta probabilidad de ser real. Además, como se verá, la igualdad de dominios no es necesaria, sólo es necesario contener un dominio en otro.

El ejemplo siguiente puede sugerir qué esperar en tales situaciones:
Sean las relaciones CLIENTE, ORDEN e ITEM como se definieron anteriormente. Los atributos #orden de las relaciones ORDEN e ITEM, evidentemente significan la misma cosa en ambas relaciones. Se espera que cualquier #orden que aparezca en una relación aparezca en la otra. Si sólo aparece en ORDEN sería una orden de nada, lo que no es erróneo, pero tampoco es el caso típico. Un #orden que aparezca sólo en ITEM será probablemente un error. Una situación menos obvia se da con el equijoin: CLIENTE $\bowtie_{nombre=cliente}$ ORDEN. Evidentemente, nombre y cliente se refieren al mismo dominio. Dado que nombre es la llave de CLIENTE, se espera que todos los clientes comunes aparezcan en el dominio de nombre. Quizás todos o la mayoría de los clientes han pedido órdenes en algún momento, tal que se puedan asumir idénticos los dominios de nombre y cliente. Sin embargo, también es posible que sólo una pequeña fracción de clientes hallan solicitado órdenes, tal que el dominio del cliente sea mucho más pequeño que el de nombre.

Afortunadamente, bajo estas premisas no importa cuál de ambos casos ocurre. El tamaño estimado del equijoin anterior no depende de fracción de los clientes colocados en el momento de órdenes de pedido. La razón intuitiva es que cada tupla de ORDEN tendrá un valor para cliente que es igualmente probable de aparecer en el atributo nombre de cualquier tupla de CLIENTE.

En este ejemplo: nombre es la llave de CLIENTE, entonces cada cliente debe aparecer exactamente una vez en esta relación. Así, el tamaño del join será el mismo de la relación ORDEN, sin importar que fracción de clientes ha colocado órdenes.

Implicaciones del modelo respecto al tamaño de salida del join:

Relación AB (T_{AB} número de tuplas.)

Atributo A	Atributo B
D_A Dominio	E_B Dominio
I_A Tamaño	J_B Tamaño
a Valor	b Valor

Relación BC (T_{BC} número de tuplas)

Atributo B	Atributo C
D_B Dominio	D_C Dominio
I_B Tamaño	I_C Tamaño
b Valor	c Valor

$E_B \subseteq D_B$.

Generalizando la observación del ejemplo anterior respecto al equijoin, considerando la reunión natural AB $\bowtie$ BC, donde existe una dependencia de inclusión en los atributos de B; específicamente, todo valor-b de la relación AB también aparece en el atributo B de por lo menos una tupla de BC, pero no necesariamente viceversa.

Se definen D_A, D_B –y como D_C los dominios de A en AB, B en BC, y C en BC respectivamente.

Se usa E_B para el dominio de B en AB; note que $E_B \subseteq D_B$.

Se usa I_A, I_B, e I_C para el tamaño de la imagen de los atributos A,B,C, esto es, el tamaño de D_A, D_B , y D_C,respectivamente.

Se usa J_B para el tamaño de la imagen de E_B.

Sea T_{AB} el número de tuplas de AB y T_{BC} las tuplas de BC.

Para calcular el tamaño de AB ⋈ BC: considere una tupla arbitraria abc en D_A x D_B x D_C donde el número posible de tales tuplas es $I_A \cdot I_B \cdot I_C$. Se necesita sólo calcular la probabilidad de que abc esté en el join, para ello ab debe estar en AB y bc en BC.

¿Cuál es la probabilidad de que una tupla aleatoria bc esté en BC?

Dadas estas premisas, esa probabilidad es: $\frac{T_{BC}}{I_B \, I_C}$

Ahora se debe hallar la probabilidad de que ab esté en AB.

a. Si b es un valor que está en D_B pero no en E_B, entonces la probabilidad será cero; este caso ocurre una fracción $(I_B - J_B) / I_B$ del tiempo.

b. Si b es un valor tal que $b \in E_B$ y $b \in D_B$, entonces la probabilidad de que ab esté en AB es: $T_{AB} / (I_A \, J_B)$, lo que ocurre cada J_B / I_B del tiempo.

Luego, la probabilidad de que una tupla aleatoria ab, escogida desde los dominios D_A x D_B, esté en AB es:

$$(\frac{I_B - J_B}{I_B} \times 0) + (\frac{J_B}{I_B} \times \frac{T_{AB}}{I_A J_B})$$

Simplificando: $\frac{T_{AB}}{I_A I_B}$

Nóte que la expresión de la probabilidad de que una tupla aleatoria ab, escogida desde los dominios D_A x D_B, esté en AB no depende de J_B. Esta observación es la formalización del comentario intuitivo de que el tamaño del join no depende de cuantos clientes han solicitado órdenes en el momento de su ejecución en el ejemplo anterior.

Multiplicando la probabilidad de encontrar ab en AB, por la probabilidad de hallar bc en BC, se calcula la probabilidad de abc en el join AB ⋈ BC:

$$\frac{T_{AB} T_{BC}}{I_A I^2{}_B I_C}$$

Finalmente, se debe multiplicar esta probabilidad por el número de tuplas posibles de hacer parte de los joins $I_A \, I_B \, I_C$, para obtener:

tamaño estimado de AB ⋈ BC = $T_{AB} \, T_{BC} / I_B$.

Esto es, el tamaño de salida es el producto del tamaño de las dos relaciones, dividido por el tamaño de la imagen del atributo común. Lo anterior se puede generalizar en el teorema siguiente.

TEOREMA: Sean $R(A_1...A_J, B_1...B_K)$ y $S(B_1...B_K, C_1...C_M)$ dos relaciones; suponer que para cada i = 1,2,...,k, el dominio de B_i en R es un subconjunto del dominio de B_i en S, o viceversa. Se asume que cada tupla en el producto de los dominios de todos los atributos de R, es igualmente probable de estar en R y similarmente para la relación S. Entonces el tamaño esperado de la reunión natural $R \bowtie S$ es:

$$\frac{T_R T_S}{I_{B_1} I_{B_2} ... I_{B_k}}$$

Donde I_{B_i} es el tamaño del dominio de B_i en R o S y donde uno es un superconjunto del otro. Se puede ahora estimar el número esperado de bloques ocupados por la salida del join. Sea I el denominador de la expresión anterior, esto es, el producto del tamaño del dominio para cada atributo compartido por R y S. Entonces 1 / I parte de las tuplas del producto R X S aparecerán en el join, tal que el tamaño de salida es:

$$\frac{B_R T_S + T_R B_S}{I}$$

Sin embargo, recuerde que si se toma una reunión natural en lugar de un equijoin, copias duplicadas del atributo del join son borradas de todas las tuplas. Luego, el estimado anterior ($\frac{B_R T_S + T_R B_S}{I}$) es ligeramente más alto, pues las tuplas no son tan grandes en la salida del join como en el producto o en el equijoin.

Ejemplo: Considerar las relaciones R(A, B, C) y S(B, C, D). Suponga que se mantiene la dependencia de inclusión $R.B \subseteq S.B$ y $S.C \subseteq R.C$. Finalmente, suponga que el tamaño del dominio para B en S es I_B =50, y el tamaño del dominio para C en R es I_C =40. Los demás parámetros para R y S son: R = 5 000 tuplas acomodadas en 500 bloques y S = 1 000 tuplas acomodadas en 100 bloques. Entonces el tamaño del join $R \bowtie S$ está dado por:

$$U = \frac{B_R T_S + B_S T_R}{I_B I_C} = \frac{500 \times 1000 + 100 \times 5000}{50 \times 40} = 500$$

Esta conclusión, que solo uno en 2 000 pares de tuplas de R y S se enlazan exitosamente, es razonable bajo este modelo de relación. Un par de tuplas tienen una opción en 50 de coincidir en B y una opción de uno en 40 de coincidir en C. Sin embargo, se debe ser muy cuidadoso si B y C no son independientes. Como un ejemplo extremo se puede suponer que existe una dependencia funcional $B \rightarrow C$ que se cumple en ambas relaciones R y S, es decir, tuplas que coincidan en B deben coincidir en C, cuando las tuplas vengan desde R, desde S ó desde ambas relaciones. Luego, tuplas que coincidan en B con seguridad coinciden en C, y la probabilidad de que dos tuplas provenientes de R y S coincidan es 1/50, no 1/2 000. Puesto de otra manera, se pudiera esperar un join de tamaño 20.000 bloques, en lugar de los 500 bloques calculados anteriormente.

La si11tuación anterior puede ser acomodada en este modelo si se define que B y C puedan ser tratados como un solo atributo llamado E. La dependencia de inclusión $R.B \subseteq S.B$, al igual que la aseveración de la dependencia funcional $B \rightarrow C$, que se mantiene en la combinación de R y S, implican que $R.E \subseteq S.E$. El tamaño del dominio de E en S es evidentemente 50, porque cada uno de los 50 valores en el dominio de B está asociado con únicamente un valor de C (el hecho de que varios valores de B puedan aparecer con un valor de C no afecta el tamaño del dominio de E). Así, se puede manejar el join $R \bowtie S$ como $AE \bowtie ED$, y calcular el tamaño de salida como 1.000.000 dividido por I_E, o 20.000.

Métodos para calcular join

Se debe considerar y evaluar el costo de varios métodos para computar join, basados en el modelo anteriormente. Los métodos considerados incluyen: la obvia opción de selección-en-un-producto, una técnica llamada join-clasificado (*sort-join*) donde ambas relaciones están clasificadas por los atributos del join, y métodos que toman ventaja de varios tipos de índices. Las fórmulas para el tiempo esperado de ejecución de estos algoritmos son frecuentemente complicadas.

Cómputo del join por selección en un producto

El modo obvio para computar R⋈S es calcular R X S (antes se asumió que S es la relación más pequeña). Sin embargo, en lugar de emitir toda tupla uv, donde u está en R y v en S, se emiten sólo esa tuplas que coinciden en los atributos comunes de R y S.

El costo de este método es el costo de entrada más el costo de salida. El lector puede chequear la fórmula siguiente de la suma, que es el costo total para llevar a cabo un join por el método de selección-en-un-producto.

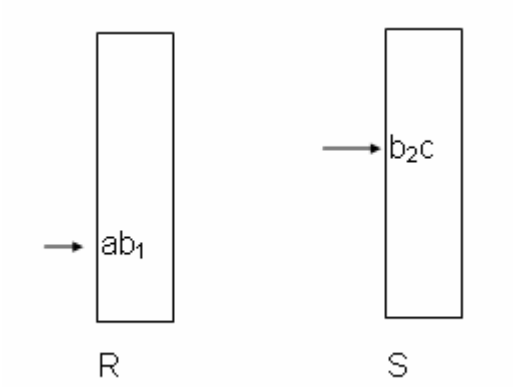
$$B_R(\frac{T_S}{I} + \frac{B_S}{M-1}) + B_S(\frac{T_R}{I} + 1)$$

Join-clasificado (*Sort-Join*):

El teorema enunciado anteriormente plantea que no se pueden llevar a cabo mejoras más significativas que la fórmula $B_R(\frac{T_S}{I} + \frac{B_S}{M-1}) + B_S(\frac{T_R}{I} + 1)$, si se ejecuta un join de tal manera que cada uno de los bloques de una relación encuentre a cada uno de los bloques de la otra en memoria principal. Sin embargo, existen varias alternativas para preprocesar una relación o tomar ventaja de los índices existentes, y así evitar la mezcla no selectiva de dos relaciones. A continuación se considera uno de esos casos, el join clasificado.

En lo que sigue, se asume que las dos relaciones para join son R(A, B) y S(B, C). La idea se generaliza fácilmente al caso donde hay varios atributos en común y/o varios atributos pertenecientes sólo a una de las relaciones. Se comienza clasificando a R por su atributo B, y se procede de la misma manera con S. Cuando se usa un par de cursores para que recorran los atributos de R y S, primero se hallan los valores-b siguiente como sugiere la figura más abajo. Suponga que se buscan dos tuplas ab_1

desde R y b_2 c desde S. Si entre b_1 y b_2 alguno es más pequeño que el otro, se avanza el cursor del más pequeño hacia abajo y se deja fijo al otro. En este caso no es posible que el más pequeño coincida en algún momento con un valor de la otra relación. Si $b_1 = b_2$, entonces se hallan todas las tuplas siguientes de R y S con el mismo valor-b. Se sigue apareando cada una de las tuplas de R con cada una de las tuplas de S y emitiendo las tuplas resultantes (borrando una copia del valor-b común). Luego, se mueve el cursor de las dos relaciones justo después de la última tupla con el valor-b en proceso.



Un ejemplo de relaciones clasificadas es:

R	S
a_1b_1	b_1c_1
a_2b_1	b_1c_2
a_3b_2	b_3c_3
a_4b_3	

Ejemplo: suponga que R y S consisten de la lista de tuplas de la tabla ejemplo de relaciones clasificadas que aparece arriba. Ambas relaciones ya han sido clasificadas por el valor-b. Se comienza con a_1b_1 para R y b_1c_1 para S. Como los valores-b coinciden, se hallan todas las tuplas de R con este valor-b. Esto es, {a_1b_1, a_2b_1}, y todas las tuplas de S con este mismo valor-b, {b_1c_1, b_1c_2}. Note que las tuplas deseadas deben estar en posiciones consecutivas en los cursores de las respectivas relaciones. Se toma el producto y se omite una copia de b_1 para cada tupla

resultante, consiguiendo entonces cuatro tuplas, {$a_1b_1c_1$, $a_1b_1c_2$, $a_2b_1c_1$, $a_2b_1c_2$}, que pertenecen a R⋈S.

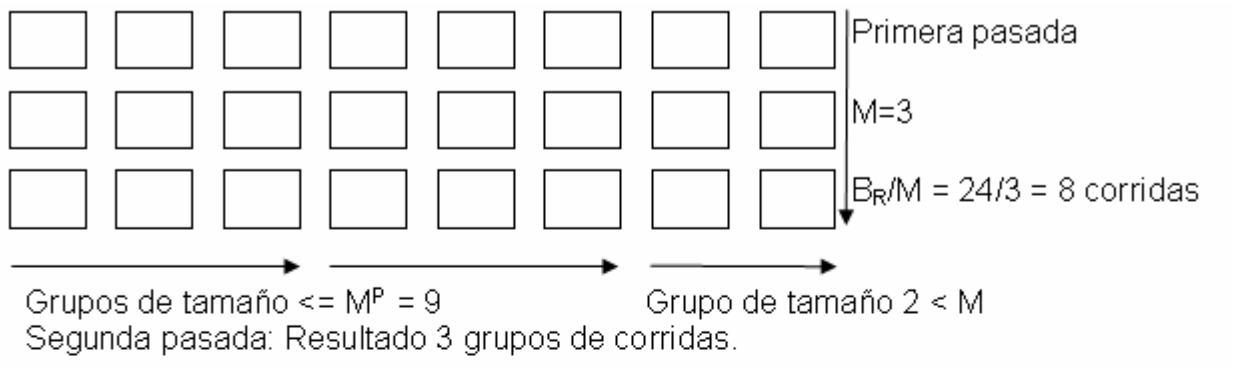
A continuación se avanza el cursor de R a a_3b_2 y el cursor de S a b_3c_3. Como b_2, presumiblemente, es menor en orden de clasificación que b_3, se sabe que no hay tuplas de S con valor-b b_2. Si las hubiese, deberían preceder a b_3. Así, se avanza el cursor de R a a_4b_3 y se deja fijo al cursor de S. Como resultado se obtienen ambos cursores en el valor-b b_3, y en cada uno de los casos, las tuplas recorridas son únicamente las tuplas de las relaciones con ese valor-b. Así, simplemente se aparean para generar la quinta y última tupla del join, {$a_4b_3c_3$}. En este punto se han recorrido ambos cursores hasta el final. En general, en el recorrido se pudiera terminar con un cursor mientras que el otro todavía tenga tuplas por recorrer.

Clasificación-mezcla multimodo (*Multiway Merge-Sort*)

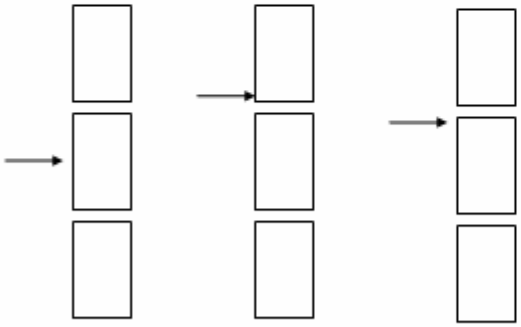
Dado que ordinariamente las relaciones no están clasificadas, se debe calcular el costo de clasificar una relación. Bajo este modelo de costos, donde se cuentan los bloques accesados, un método llamado clasificación-mezcla multimodo es una excelente alternativa. El lector interesado en los detalles de este algoritmo puede consultar Aho, Hopcroft y Ullman (1983). Aquí sólo se describe la idea general y se da un análisis de costo. El algoritmo opera construyendo corridas, que no son otra cosa que secuencias de bloques que contienen una lista de tuplas clasificadas y pertenecientes a una relación R dada. En una serie de pasadas, R es particionada en un conjunto de corridas, progresivamente más grandes, hasta el final, cuando solo queda una corrida, con la relación completa. Se asume que hay M bloques de memoria principal disponibles para guardar bloques de R. Algunos bloques adicionales de memoria son requeridos para cálculos, tal como hallar el más pequeño de un conjunto de M valores, y para la salida clasificada. También se asume que ya están reservados y que no forman parte de los M bloques disponibles para almacenar la relación a ser clasificada.

En la primera pasada, se leen los bloques de R a memoria principal, M a la vez, y se clasifican las tuplas dentro del grupo de M bloques, usando cualquier algoritmo de clasificación interna apropiado, tal como el Quick-Sort (véase Aho, Hopcroft y Ullman, 1983). Note que no se toman en cuenta los cálculos en memoria.

En realidad, la clasificación de M bloques toma probablemente mucho menos tiempo que su lectura desde almacenamiento secundario. Como resultado, R está ahora particionado en B_R/M corridas de M bloques cada una. En la pasada p, con p > 1, se inicia con corridas de M^{P-1} bloques cada una, y se reunen en grupos de M corridas. Así, cada nuevo grupo consiste de M^P bloques. El último grupo puede tener un poco menos de M corridas, y por ello su longitud será menor que M^P bloques. Luego se mezcla cada grupo con el proceso sugerido en la figura siguiente:



Segunda pasada: Resultado 3 grupos de corridas.



Mezcla de un grupo.

Mezcla Multimodo con M=3 y p=2.

Para ejecutar la mezcla se mantiene un cursor para cada una de las M corridas en un grupo. Al comienzo, cada cursor está apuntando al elemento más pequeño al inicio de la corrida. Repetidamente se selecciona el elemento más pequeño entre los señalados por los M cursores, y el cursor cuyo elemento sea escogido es avanzado al elemento siguiente de esa corrida. El elemento seleccionado es adicionado al bloque de salida actual. Se necesita mantener en memoria principal sólo un bloque de cada corrida. Cuando el cursor para una corrida termina de recorrer tal bloque, es reemplazado por el bloque siguiente de la misma corrida. Igualmente, se necesita mantener en memoria principal sólo un bloque para la salida, que no ha sido contabilizado entre los M reservados para la relación de entrada. Cuando un bloque es llenado, es movido a almacenamiento secundario, por lo que puede ser llenado de nuevo con las tuplas siguiente en el orden de clasificación. Después de un número de pasadas igual a $\lceil \log_M(B_R) \rceil$, menor entero por encima de $\log_M(B_R)$, las corridas producidas son por lo menos de longitud B_R. Así, la relación completa R es una corrida. Esto es, está clasificada (Si $B_R/M^P = 1$, la relación está clasificada).

Análisis del join-clasificado (*sort-join*)

Se estima el costo de ejecutar un join ordenado entre las relaciones R y S. El proceso final del join tiene como costo de entrada $B_R + B_S$, dado que se necesita leer cada bloque de las relaciones clasificadas solo una vez. El costo de salida es determinado obviamente por $\frac{B_R T_S + T_R B_S}{I}$.

Ahora se debe estimar el costo de clasificar las relaciones. Una pasada de la clasificación-mezcla multimodo lee cada bloque de la relación una vez, pues se lee cada corrida solo una vez. También escribe nuevos bloques, que almacenan mezcladas las corridas de cada grupo de la relación. Así, cada pasada usa 2B accesos a bloques, si la relación requiere de B bloques para almacenarse de una manera compacta. Como el número de pasadas es $\lceil \log_M(B) \rceil$, el costo de usar el algoritmo en cuestión en una relación de B bloques es $2B \times \log_M B$. Así, el costo total del join es:

$$2B_R \log_M B_R + 2B_S \log_M B_S + B_R + B_S + \frac{B_R T_S + T_R B_S}{I}$$

Los dos primeros términos son el costo de clasificar las relaciones, los siguientes dos son el costo de recorrer las relaciones clasificadas para aparear las tuplas coincidentes y el último término es el costo de la salida. Esta fórmula es válida siempre que no existan valores-b de los atributos de enlace para los que el número de tuplas que comparten tales valores en las dos relaciones excedan el número de ellas que se pueden acomodar al mismo tiempo en memoria principal. Si lo anterior no se cumple, entonces el recorrido de las relaciones sugerido por la figura de la página 111 no podría llevarse a cabo sin la relectura de algunos de los bloques que contengan tales valores-b. En general, si el tamaño de B_R y B_S, bloques ocupados por las relaciones, es grande, con M constante y $B_R >= B_S$, el costo de entrada para la selección en un producto crecerá tanto como $\frac{B_R B_S}{M}$. Mientras que el costo de entrada para el join-clasificado crecerá tanto como $B_R \log_M B_R$, cifra significativamente menor.

Join usando un índice

Considere de nuevo el join R(A,B)⋈S(B,C), pero ahora suponga que S tiene un índice cluster en B. Asuma que R tiene un almacenamiento compacto. Entonces se puede ejecutar el join tomando cada bloque de R, y para cada tupla de R, usamos el índice para hallar las tuplas de S que coinciden. Esto es, hacer:

FOR cada bloque β de R DO
FOR cada tupla ab en el bloque β DO
join ab con cada tupla en $\sigma_{B=b}(S)$

Sea I el tamaño de la imagen de B en S y asúmase que la dependencia de inclusión R.B $\subseteq$ S.B. se conserva. Esto es, todo valor-b que aparezca en R aparece en S. Entonces, el ciclo interno de del algoritmo anterior es llevado a cabo T_R veces y en cada iteración se han recuperado, vía el índice cluster, cerca de B_S / I bloques. En el ciclo externo se lee cada bloque de R una vez, implicando un costo adicional de B_R

bloques accesados. Así, el costo de entrada es igual a $B_R + \frac{T_R B_S}{I}$, y el costo total es $B_R(1+\frac{T_S}{I}) + \frac{2T_R B_S}{I}$.

Se debe ser cuidadoso al interpretar las fórmulas, pues se ha asumido tácitamente que I < = B_S. Si no, Entonces B_S/I no es un estimado adecuado del número de bloques recuperados, pues el número no puede ser menor que 1. Si I >= B_S, entonces se recupera cerca de un bloque de S por cada tupla de R, y la fórmula $B_R + \frac{T_R B_S}{I}$ debe ser reemplazada por B_R + T_R. Una fórmula que generaliza a $B_R(1+\frac{T_S}{I}) + \frac{2T_R B_S}{I}$ para el caso donde I puede ser mayor que o menor que B_S es:

$$B_R(1+\frac{T_S}{I}) + \frac{T_R B_S}{I} + T_R \times \max(1, \frac{B_S}{I})$$

Algunos otros casos de join con índices

Se pueden derivar varias fórmulas similares para situaciones relacionadas. Si la relación R no está compacta, entonces se puede tener acceso a T_R bloques, en lugar de B_R, para leer a R. En este caso, el costo de entrada debería ser T_R (1+B_S / I), en lugar de $B_R + \frac{T_R B_S}{I}$. Por supuesto el costo de salida no cambia. Si R está compacta, pero el índice de S no es cluster, entonces cada iteración del ciclo interno del algoritmo de join usando un índice debe recuperar cerca de T_S / I bloques, y el costo de entrada debería ser B_R + $T_R T_S$ / I. Si el índice no es cluster y R no está compacta, entonces el costo de entrada es T_R(1+ T_S / I). Finalmente, se pueden derivar fórmulas similares, con R y S intercambiadas, si hay un índice de R en B que se pueda usar.

En todos los casos anteriores se ha asumido que R.B $\subseteq$ S.B. Si la dependencia de inclusión se presenta de otra manera, entonces $B_R(\frac{B_S}{M-1}) + B_S$ será todavía un buen estimado del costo de entrada, asumiendo que el tamaño de la imagen de S.B no es

mayor que B_S, pero se debe interpretar I como $I_{R.B}$., en lugar de $I_{S.B}$, aunque el índice todavía sea de S. La fórmula apropiada para el caso donde S.B $\subseteq$ R.B, permitiendo $I_{S.B}$< = B_S e $I_{S.B}$ > B_S, es $B_R(1+\frac{T_S}{J})+\frac{T_R}{J}(B_S+\max(B_S,I))$ donde J= $I_{R.B}$ e I = $I_{S.B}$

Ejemplo: tomando el ejemplo de las corridas, con B_R = 500, T_R=5000, B_S =100 y T_S = 1 000. Suponga que R.B $\subseteq$ S.B, y que I = $I_{S.B}$ = 50. Finalmente, asumiendo que hay un índice cluster en el atributo B de S. Como I < = B_S, aplicando $B_R(1+\frac{T_S}{I})+\frac{2T_R B_S}{I}$, cuyos valores son:

$$500(1+\frac{1000}{50})+\frac{2\times 5000\times 100}{50}=30.500$$

Suponga la misma situación, pero ahora S.B $\subseteq$ R.B, y J = $I_{R.B}$ = 200. Además, suponga que el índice de S.B no es cluster. Entonces el costo de entrada es:

$B_R + T_R T_S / J$ = 500 + (5000 x 1000) / 200 = 25.500

Mientras que el costo de salida permanece constante, esto es 5000. Así, el costo total se incrementa desde 8 000, si tuviese índice cluster, a 30.500.

Join usando dos índices:

Aún se puede mejorar el costo si hay un índice en B para ambas relaciones. Suponga primero que ambos son índices cluster. Se puede hallar a un conjunto de valores-b examinando uno de los índices. Se usa para ello al índice con el tamaño de imagen más pequeña. Se asume que es el índice de B en S, y que sea I = $I_{B.S}$. Como se accede al índice de B en S, se hallan todos los I valores del B en turno. Note que es innecesario hallar al conjunto de valores-b en R, porque si están ausentes en S, de todas maneras no aparecerán en el resultado final del join.

Una vez que se tenga el conjunto de valores-b, se pueden recorrer, recuperando las tuplas relevantes de S y R. Formalmente:

FOR cada valor-b b DO

join las tuplas de $\sigma_{B=b}(R)$ con $\sigma_{B=b}(S)$

Análisis del join con dos índices

Se estima el costo del algoritmo para el join usando índices, asumiendo S.B $\subseteq$ R.B. El cuerpo del mismo es ejecutado I veces. El número promedio de bloques recuperados en una iteración es a lo sumo el max (1, B_R / J), donde J es $I_{R.B}$. Podría ser menor si muchos valores en el dominio de B en S no están en el dominio de B en R. El número de bloques de S recuperados es cerca del max(1, B_S / I). Así, el costo de entrada para este método de join es a lo sumo

$$I \times (\max(1, \frac{B_R}{J}) + \max(1, \frac{B_S}{I}))$$

Cuando se calcula el costo de salida con $\frac{B_R T_S + T_R B_S}{I}$, se debe recordar que el tamaño de la imagen es ellmás grande entre las dos imagenes de los atributos usados para el join. Así, I es J para este caso, y el costo total del join con dos indices es:

$$I \times \max(1, \frac{B_R}{J}) + \max(I, B_S) + \frac{B_R T_S + T_R B_S}{J}$$

Queda como propuesta al lector, las modificaciones necesarias a la expresión anterior para cuando ambos índices no sean del tipo cluster.

Creando un índice cluster

Como parece tener considerable ventaja ejecutar los join con índices cluster en cada uno de los atributos involucrados en la condición, considere qué tan rápido se pudiera crear tal índice si no existe. Suponga que se tiene una relación R(A,B), y que sea I el tamaño de la imagen para B, el atributo en el que se va a crear el índice cluster. Lo que se hace es crear una tabla aleatoria con cerca de I bloques, y en cada bloque se ubican copias de todas las tuplas de R cuyo valor-b le corresponde al aplicar la función aleatoria. Como promedio se tendrá un valor-b por bloque. Suponga que se fuerza a usar sólo M bloques de memoria principal para guardar el

contenido de los bloques que se conformen. Como en el análisis del join-clasificado, no se cuentan otros requerimientos de memoria entre estos M bloques, tal como espacio para almacenar el programa y un bloque para lecturas de entrada. Se crean en varios pasos la tabla aleatoria con I bloques, análogos a los pasos usados para la clasificación-mezcla multimodo. Seleccione una función aleatoria que disperse las tuplas en los I bloques numerados de 0 a I-1. Si I > M, no se puede particionar de una vez a R en todos los I bloques, porque se gastarían tantos accesos a bloques, moviendolos varias veces en los bloques a y desde la memoria, como tuplas de R fuesen ubicadas por la función hash en cada bloque. Sin embargo, se puede dividir a R en M "superbloques", cada uno representando I/M de los bloques originales. Separando un bloque para cada superbloque en memoria, y cuando esté lleno, se mueve a memoria secundaria y se inicia un nuevo bloque para el mismo superbloque. En la primera pasada se usa el superbloque 0 para representar los bloques 0 hasta (I/M) -1, el superbloque 1 para representar I/M hasta (2I/M)-1, y así sucesivamente. Se lee cada bloque de R, se aplica la función a cada tupla del bloque, y si la tupla pertenece al bloque i, se coloca en el superbloque $\lfloor iM/I \rfloor$.

En general, en el paso p, con p >1, se inicia con superbloques donde cada uno representa I/M^{P-1} bloques originales y se finaliza con un número mayor de superbloques de menor tamaño, cada una representando I/M^{P} bloques originales. Durante estas pasadas se trabaja en un superbloque a la vez, dividiéndolo para representar M grupos de bloques. Cada grupo pertenecerá a uno de los nuevos superbloques. Se lee cada viejo superbloque una vez por pasada, y se guardan M bloques representando los M nuevos superbloques más pequeños, escribiendo a disco el bloque cuando se llene. Despues de $\log_M I$ pasadas cada nuevo superbloque representa solo un bloque de los I originales, y el proceso termina.

Análisis del join con creación de índices

El número de bloques accesados que requiere la técnica anterior para la creación del índice se calcula de la manera siguiente. Cada pasada lee y escribe cerca de B_R bloques, son leídos todos los bloques de los superbloques en que se divide a R y son

escritos todos los superbloques con un tamaño más pequeño. Así, el costo de las $\log_M$I pasadas es: $2B_R\log_M$I. Esta fórmula será una subestimación si I > B_R. Porque entonces, la pasada o pasadas finales dividirán a R en más bloques que las necesarias para almacenarla compacta. Por lo tanto, estas pasadas usarán más de $2B_R$ accesos a bloques. Un límite superior en el costo, sin asumir I > = B_R es $2\max(I,B_R)\log_M I$. Se deja como ejercicio encontrar una fórmula más precisa.

Suponga que se desea obtener el join R(A, B)⋈S(B, C), pero no hay índices en B para las relaciones. Se hace uso de la técnica ya descrita para crear tales índices, pagando para cada creación la cantidad extra calculada por $2\max(I,B_R)\log_M I$. Asumir que $I_{R.B}$ es denotado por J e $I_{S.B}$ es denotado por I. También se asume la dependencia de inclusión R.B $\subseteq$ S.B, con la consecuencia de J > = I. Entonces, el costo de este método es $I\times\max(1,\frac{B_R}{J})+\max(I,B_S)+\frac{B_R T_S + T_R B_S}{J}$ más el costo de crear los índices de R y S. El último costo está dado por $2\max(I,B_R)\log_M I$, con la apropiada sustitución de las variables. La fórmula para el costo total es:

$$(I+2J\times\log_M J)\times\max(1,\frac{B_R}{J})+\max(I,B_S)(1+2\times\log_M I)+\frac{B_R T_S + T_R B_S}{J}\ .$$

Resumen y comparaciones

La tabla que aparece a continuación es un resumen de los principales métodos de join considerados en este material. En adición a las premisas ya descritas explícitamente, se asumen las premisas siguientes y convenciones globales.

1. Se cumple el modelo de relaciones aleatorias.
2. Las relaciones se almacenan de una manera compacta.
3. El join es R(A,B)⋈S(B,C). R y S tienen T_R y T_S tuplas, respectivamente, y se acomodan en B_R y B_S bloques.

4. I es $I_{S.B}$, el tamaño de la imagen de B en S, y J es $I_{R.B}$, el tamaño de la imagen de B en R.

5. M es aproximadamente el tamaño de la memoria principal, excluyendo el espacio necesario para almacenar los programas, y en algunos casos, un buffer de entrada o salida.

A continuación se resumen los métodos de join. Los costos se listan debajo de la tabla debido a la extensión de las fórmulas.

Metodo	Premisas	Costo	Término dominante
Selección-en-un-producto.	$R.B \subseteq S.B$ o $S.B \subseteq R.B$	C1	$B_R B_S / M$
Join-clasificado	$R.B \subseteq S.B$ $B_R / J + B_S / I <= M$	C2	$B_R \log B_R$, $B_S \log B_S$ $B_R T_S / I$, $B_S T_R / I$
Índice cluster en S.B	$R.B \subseteq S.B$	C3	$B_R T_S / I$, $B_S T_R / I$, T_R
Índice cluster en S.B.	$S.B \subseteq R.B$	C4	B_R , $B_R T_S / J$, $B_S T_R / J$ $T_R I / J$
Join con dos índices.	$S.B \subseteq R.B$ $B_R / J + B_S / I <= M$	C5	I, $B_R T_S / J$, $B_S T_R / J$
Join con creación de los dos índices	$S.B \subseteq R.B$ $B_R / J + B_S / I <= M$	C6	$J \log_M J$, $B_R \log_M J$, $B_S \log_M I$, $B_R T_S / J$, $B_S T_R / J$

C1. $B_R(\frac{T_S}{I} + \frac{B_S}{M-1}) + B_S(\frac{T_R}{I} + 1)$

C2. $2B_R \log_M B_R + 2B_S \log_M B_S + B_R + B_S + \frac{B_R T_S + T_R B_S}{I}$

C3. $B_R(1 + \frac{T_S}{I}) + \frac{T_R B_S}{I} + T_R \times \max(1, \frac{B_S}{I})$

C4. $B_R(1 + \frac{T_S}{J}) + \frac{T_R}{J}(B_S + \max(B_S, I))$

C5. $I \times \max(1, \frac{B_R}{J}) + \max(I, B_S) + \frac{B_R T_S + T_R B_S}{J}$

C6. $(I + 2J \times \log_M J) \times \max(1, \frac{B_R}{J}) + \max(I, B_S)(1 + 2 \times \log_M I) + \frac{B_R T_S + T_R B_S}{J}$

También se sabe que I < = T_S, y que J < = T_R. Además, si se asume la dependencia de inclusión S.B $\subseteq$ R.B, entonces se sabe que I < = J. Si se asume la inclusión opuesta, entonces I > = J. Se asumen estas desigualdades para eliminar ciertos términos de las fórmulas que son, por ende, más pequeños que otros términos de la misma fórmula. La segunda columna de la tabla da las premisas especiales necesarias para hallar las fórmulas de costos válidas, dadas por el número de ecuación de costos de la tercera columna. Sin embargo, se pueden distinguir dos tipos de premisas. La dependencia de inclusión entre los dominios B en R y S es necesaria para estimar el tamaño de salida. Si no se conservan las dependencias de inclusión, entonces, como promedio, las salidas serán más pequeñas, tal que las fórmulas de costo son límites superiores al costo verdadero. El segundo tipo de premisa establece que el tamaño de la memoria principal debe ser lo suficientemente grande para almacenar a todas las tuplas de R y S que compartan un valor-b común, esto es, B_R /J + B_S /I < = M.

Si la relación no se mantiene, entonces ciertas operaciones que fueron asumidas, se llevan a cabo en memoria principal, requieren accesos adicionales a bloques para llevarlos y sacarlos desde memoria principal, como se hizo en el algoritmo del producto tratado anteriormente. Así, si estas condiciones son violadas, el costo dado en la columna 3 será un estimado bajo. La última columna de la tabla resumida, lista los términos dominantes. Para hallar el término dominante de cada fórmula, se asume que B_R, B_S, T_R, T_S, I y J crecen en proporción, pero M permanece constante. Las fórmulas fueron expandidas y cualquier término que no fuera asintóticamente mayor que otro fue eliminado, dejando los términos dominantes. Por supuesto, en dependencia de los valores de los parámetros, alguno de los términos dominantes para una fórmula podría resultar ser el más grande.

El término dominante dice cuál método es el mejor a medida que crese la relación y el tamaño de la memoria principal permanece fija. En particular, el único término dominante, que es cuadrático respecto a la relación, es $B_R B_S$ / M del join con selección en un producto. Otros términos, o crecen como el producto del tamaño de la relación por el logaritmo de este tamaño, por ejemplo $B_R \log_M J$ del join con creación de índices, o crecen linealmente con el tamaño de la relación, por ejemplo, $B_R T_S$ / I. Note cómo se asume que I, el tamaño de la imagen de R.B., crece en proporción a B_R, el número de bloques tomados por la relación R. Así, para grandes relaciones, se espera un join con selección en un producto como el peor desempeño, aunque puede ser eficiente en pequeñas relaciones dada su simplicidad. La selección en un producto lee repetidamente un bloque de R y luego compara sus tuplas con todas las tuplas de los M-1 bloques de S. Este cómputo en memoria principal puede significar costoso si M es grande. El join que usa un índice es el de mejor desempeño, pero no es una opción si no existe tal índice. Entonces, las alternativas válidas son un join-clasificado o la creación de un índice. El valor particular de los parámetros determina cuál es mejor en cada situación dada.

OPTIMIZACIÓN POR MANIPULACIÓN ALGEBRAICA

Hasta ahora, en este capítulo, se han comparado estrategias para ejecutar las operaciones relacionales básicas: selección, producto y join. Ahora se dará un vistazo al problema completo de afinar una consulta, escrita en algún lenguaje relacional, en una secuencia de pasos algebraicos relacionales que juntos forman un algoritmo eficiente para responderla.

La primera cosa que un optimizador de consultas lleva a cabo es convertir la consulta en una forma interna que se asemeje al algebra relacional. Por ejemplo, el procesador de consultas QUEL, usado por INGRES, donde un ejemplo de consulta puede ser: RANGE OF e IS empleado
RANGE OF d IS departamento

RETRIEVE (e.nombre, d.nombre) WHERE e.deptno = d.deptno;

Y cuyo algoritmo de optimización se discute más adelante, comienza por asumir un producto cartesiano de relaciones $R_1xR_2x...x\ R_K$, si los rangos aplicables a las declaraciones son de la forma Range Of t_i IS R_i, para i = 1, 2,..., k. Luego, la cláusula WHERE de la consulta QUEL es reemplazada por una selección y los componentes mencionados en la cláusula retrieve son obtenidos por una proyección. En realidad, las formas básicas de consulta del Query-by-Example y del SQL son también reemplazadas por expresiones algebraicas similares, una proyección en una selección en un producto.

Equivalencia de expresiones

Antes de poder “optimizar” expresiones se debe entender claramente cuándo dos expresiones del álgebra relacional (de ahora en adelante referidas como expresiones relacionales) son equivalentes. Primero se debe recordar que hay dos definiciones de relación en uso y que ellas tienen propiedades matemáticas diferentes. El primer punto de vista es que una relación es un conjunto de k-tuplas para un k fijo, y dos relaciones son iguales si y sólo si tienen el mismo conjunto de tuplas. El segundo punto de vista define a una relación como un conjunto de mapeos, desde un conjunto de nombres de atributos a valores. Dos relaciones son consideradas iguales si tienen el mismo conjunto de mapeos. Una relación en el primer sentido puede ser convertida a una relación en el segundo sentido dando nombre de atributo a las columnas. Se puede convertir desde la segunda definición a la primera estableciendo un orden fijo para los atributos.

Aquí se usa la segunda definición, una relación es un conjunto de mapeos desde atributos a valores. La justificación está en que todos los lenguajes de consulta existentes permiten, y generalmente requieren, nombres para las columnas de una relación. De mayor importancia, el orden en el cual las columnas de la tabla son impresas no es significativo, en tanto cada columna está etiquetada con su propio nombre de atributo. Donde sea posible, se adoptan nombres para cada atributo de la

relación computada en una expresión relacional de los nombres de atributos para los argumentos de la expresión. También se requiere que los nombres sean aportados por el resultado de una unión o diferencia de conjuntos.

Una expresión relacional cuyos operandos son relaciones R_1, R_2,..., R_K , define una función cuyo dominio son k-tuplas de las relaciones (r_1,r_2,...,r_k), donde cada r_i es una relación de la aridad apropiada a R_i. El valor de la función es la relación que resulta cuando se sustituye cada r_i por R_i y luego se evalúa la expresión. Dos expresiones E_1 y E_2 son equivalentes, escrito $E_1 \equiv E_2$, si ellas representan el mismo mapeo. Esto es, cuando se sustituye la misma relación por nombres idénticos en las dos expresiones, se obtiene el mismo resultado. Con esta definición de equivalencia se pueden listar algunas transformaciones algebraicas útiles.

Leyes que involucran join y productos cartesianos: En el teorema 2.2 de la sección 2.4 del volumen I de Ullman (1989), se prueba cómo la reunión natural (join natural) es conmutativa y asociativa. Leyes algebraicas similares se cumplen para los θ-join y para los productos. A continuación se establecen las mismas:

1. Leyes conmutativas para los join y los productos. Si E_1 y E_2 son expresiones relacionales, y F es una condición en los atributos de E_1 y E_2, entonces:

$E_1 \bowtie_F E_2 \equiv E_2 \bowtie_F E_1$

$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$

$E_1 x E_2 \equiv E_2 x E_1$

2. Leyes asociativas para join y productos. Si E_1, E_2 y E_3 son expresiones relacionales, y F_1 y F_2 son condiciones, entonces:

$(E_1 \bowtie_{F1} E_2) \bowtie_{F2} E_3 \equiv E_1 \bowtie_{F1} (E_2 \bowtie_{F2} E_3)$

$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

$(E_1 x E_2) x E_3 \equiv E_1 x (E_2 x E_3)$

El lector puede estar sorprendido por algunas de estas leyes. Por ejemplo, el producto no es conmutativo. La diferencia radica en que aquí se usa la definición de conjunto de mapeos para las relaciones, mientras que en Ullman (1989) se usa la definición de conjunto de listas. Por ejemplo, $E_1 \times E_2 \equiv E_2 \times E_1$ se conserva. En primer lugar, note que se hace una distinción de los atributos de las dos relaciones, aun si estos atributos tienen el mismo nombre. Así, sean R y S las relaciones que tienen los valores de E_1 y E_2, respectivamente. Entonces, el atributo A de R es llamado R.A en el producto de E_1 y E_2, mientras que el atributo A de S es llamado S.A. Se puede reemplazar a R.A. o S.A por A, si A es solo un atributo de R o de S, pero no de ambas.

Sea ρ una tupla de $E_1 \times E_2$. Entonces, hay una tupla μ en R y una tupla ν en S, tal que para todos los atributos A de R, $\rho[R.A] = \mu[A]$, y para todos los atributos A de S, $\rho[S.A] = \nu[A]$. Ahora, considere el producto $E_2 \times E_1$, que también debe contener una tupla τ formada desde μ y ν. Evidentemente, $\tau[R.A] = \mu[A]$ y $\tau[S.A] = \nu[A]$ para todos los atributos A de R y S, respectivamente. Además, τ es ρ. Puesto que ρ es una tupla arbitraria, se cumple que $E_1 \times E_2 \subseteq E_2 \times E_1$. La inclusión opuesta no es menos trivial, tal que se concluye que $E_1 \times E_2$ y $E_2 \times E_1$ son la misma relación.

Leyes que involucran selecciones y proyecciones: La cascada de varias proyecciones puede ser combinada en una proyección. Se expresa este hecho por:

3. Cascada de proyecciones.

$$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(E)) \equiv \pi_{A_1,\ldots,A_n}(E)$$

Note que los nombres de atributos $A_1,\ldots A_n$ deben estar entre los B_i para que la cascada tenga sentido.

Similarmente, la cascada de selecciones puede ser combinada en una selección que chequea de una vez todas las condiciones. La equivalencia siguiente permite combinar o descomponer arbitrariamente cualquier secuencia de selecciones.

4. Cascada de selecciones.

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$$

Dado que $F_1 \Lambda F_2 = F_2 \Lambda F_1$, se sigue inmediatamente que la selección puede ser conmutada, esto es,

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_2}(\sigma_{F_1}(E))$$

5. Conmutando selecciones y proyecciones. Si la condición F involucra únicamente atributos $A_1,....A_n$, entonces

$$\pi_{A_1,...,A_n}(\sigma_F(E)) \equiv \sigma_F(\pi_{A_1,...,A_n}(E))$$

Más generalmente, si la condición F también involucra atributos $B_1,...B_m$ que no están entre $A_1,...A_n$, entonces

$$\pi_{A_1,...,A_n}(\sigma_F(E)) \equiv \pi_{A_1,...,A_n}(\sigma_F(\pi_{A_1,...,A_n,B_1,...B_m}(E)))$$

En la equivalencia anterior puede parecer que la proyección extra a la derecha $\pi_{A_1,...,A_n,B_1,...B_m}$ es redundante. Sin embargo, esta es una sutileza de la notación.

Una proyección malgasta atributos, así como también preserva algunos. En particular, si la relación para E tiene un atributo C que no está entre las A y las B, entonces C es proyectado fuera antes de la selección en la fórmula de la derecha, mientras que a la izquierda es proyectado fuera después de la selección.

6. Conmutando selecciones con productos cartesianos. Si todos los atributos mencionados en F son atributos de E_1, entonces:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

Como un corolario útil, si F es de la forma $F_1 \Lambda F_2$, donde F_1 involucra sólo atributos de E_1 , y F_2 involucra sólo atributos de E_2 , se pueden usar las reglas 1, 4 y 6 para obtener:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

Adicionalmente, si F_1 involucra sólo atributos de E_1, pero F_2 involucra atributos de ambos, se puede establecer que:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$$

realizando parte de la selección antes del producto.

7. Conmutando selecciones con una unión. Si se tiene la expresión E = $E_1 \cup E_2$, se puede asumir que los atributos de E_1 y E_2 tienen los mismos nombres que los de E, o al menos, que hay una correspondencia dada que asocie cada atributo de E con un único atributo de E_1 y con un único atributo de E_2 . Así se puede escribir:

$$\sigma_F(E_1 \bigcup E_2) \equiv \sigma_F(E_1) \bigcup \sigma_F(E_2)$$

Si los nombres de los atributos para E_1 y/o E_2 difieren realmente de los de E, entonces la fórmula F de la derecha debe ser modificada para que use los nombres apropiados.

8. Conmutando selecciones con un conjunto diferencia.

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

Como en 7, si los nombres de atributos de E_1 y E_2 difieren, se deben reemplazar los atributos de F de la derecha por los nombres correspondientes para E_1. Note también que no es necesaria la selección de la derecha $\sigma_F(E_2)$. Puede ser reemplazada por E_2 si se desea. Sin embargo, es usualmente tan eficiente efectuar la selección como obtener el valor de la expresión E_2, porque la primera es un conjunto más pequeño que la última. Aquí no se establecen todas las leyes para llevar a cabo la selección antes que el join, dado que un join puede ser expresado siempre como un producto cartesiano seguido por una selección, y en el caso del join natural, una proyección. Las reglas para llevar a cabo una selección antes que un join se derivan de las reglas 4, 5 y 6. Las reglas para mover una proyección antes que un producto cartesiano o una unión son similares a las reglas 6 y 7. Note que no hay una manera general de mover una proyección antes que un conjunto diferencia. Sin embargo, hay una regla útil que se aplica a las reuniones naturales (join natural) y toma ventaja de la igualdad implícita entre los atributos usados en la definición de esta operación. Su generalización para los equijoin se deja como ejercicio al lector.

9. Conmutando selecciones con la reunión natural (Caso Especial): Si F es una condición que involucra sólo atributos compartidos por E_1 y E_2, entoces

$$\sigma_F(E_1 \bowtie E_2) \equiv \sigma_F(E_1) \bowtie \sigma_F(E_2)$$

Observe que no se está llevando a cabo trabajo extra ejecutando la selección bajo ambas ramas del árbol de la expresión. La selección reduce el tamaño de las relaciones E_1 y E_2, ahorrando trabajo en ambas ramas.

10. Conmutando una proyección con un producto cartesiano: Sean E_1 y E_2 dos expresiones relacionales. Sea $A_1,...,A_n$ una lista de atributos, de los cuales $B_1,...,B_m$ son atributos de E_1, y el resto de atributos $C_1,...,C_k$, son de E_2. Entonces

$$\pi_{A_1,...,A_n}(E_1 \times E_2) \equiv \pi_{B_1,...,B_m}(E_1) \times \pi_{C_1,...,C_k}(E_2)$$

11. Conmutando una proyección con una unión.

$$\pi_{A_1,...,A_n}(E_1 \cup E_2) \equiv \pi_{A_1,...,A_n}(E_1) \cup \pi_{A_1,...,A_n}(E_2)$$

Como en la regla 7, si los nombres de los atributos para E_1 y/o E_2 difieren de los de $E_1 \cup E_2$ se deben reemplazar $A_1,...,A_n$ de la derecha por los nombres apropiados.

Principios para la manipulación algebraica

Ahora que se han listado equivalencias útiles, se pueden formular reglas estableciendo la dirección preferida para aplicarlas. No hay un algoritmo que, dada una expresión, garantice producir una expresión equivalente óptima, pero los principios siguientes son generalmente útiles.

1. Ejecutar las selecciones tan pronto como sea posible. Esta transformación en las consultas, más que en cualquier otra expresión, es responsable de ahorrar orden de magnitud en tiempo de ejecución, pues tiende a convertir los resultados intermedios en pequeñas evaluaciones de múltiples pasos. Así, en las reglas 6-9, es preferible reemplazar las expresiones de la izquierda con sus equivalentes de la derecha.

2. Combinar en un join ciertas selecciones con un producto cartesiano previo. Como se ha visto, un join, especialmente un equijoin, puede ser considerablemente más barato que un producto cartesiano en las mismas relaciones. Cuando el resultado del producto cartesiano RxS es el argumento de una selección, y esa selección involucra comparaciones entre atributos de R y S, el producto es realmente un join. Note que una comparación que no involucre atributos de R o no involucre atributos de S,

puede ser colocada delante del producto y ser aplicada a S o a R, respectivamente, lo que es mejor que convertir el producto en un join.

3. Combinar secuencia de operaciones unarias. Una cascada de operaciones unarias ─selección y proyección─ pueden ser combinadas, aplicándolas en un grupo al recorrer cada tupla. Similarmente, se pueden combinar estas operaciones unarias con una operación binaria previa, si se aplica la operación unaria a cada tupla en el resultado de la operación binaria.

4. Buscar subexpresiones comunes en una expresión. Si el resultado de una subexpresión común (una expresión que aparece más de una vez) no es una relación grande, y puede leerse desde memoria secundaria en menos tiempo que el que toma su cálculo, entonces es ventajoso precalcularla una sola vez. Subexpresiones que involucren join que no pueden ser modificadas moviendo a su interior una selección, caen generalmente en esta categoría. Subexpresiones comunes aparecerán frecuentemente cuando las consultas son expresadas en términos de vistas, pues se debe sustituir la misma expresión para cada ocurrencia de la vista. Si varias consultas son parte de un programa compilado, entonces se tiene la oportunidad de buscar de una vez las subexpresiones comunes entre todas las consultas.

Un algoritmo para optimizar expresiones relacionales:

Se pueden aplicar las leyes anteriores para optimizar una expresión relacional. La expresión resultante “optimizada” está sujeta a los principios enunciados, aunque no hay una completa certeza que sea la óptima entre todas las expresiones equivalentes. Se intenta mover las selecciones y las proyecciones tan abajo del árbol de la expresión como sea posible, aunque si hay una cascada de tales operaciones se organizan en una selección seguida de una proyección. También, cuando sea posible, se agrupan las selecciones y las proyecciones con la operación binaria que las preceda, ya sea unión, producto, join, o diferencia.

Algunos casos especiales ocurren cuando una operación binaria tiene operandos que son selecciones y/o proyecciones aplicadas a las hojas del árbol. Se debe considerar cuidadosamente cómo las operaciones binarias se llevan a cabo, y en algunos casos desear incorporar la selección o proyección en la operación binaria. Por ejemplo, si la operación binaria es la unión, se pueden incorporar las selecciónes y las proyecciónes más abajo en el árbol, sin pérdida de eficiencia, y de cualquier modo copiar los operandos para formar la unión. Sin embargo, si la operación binaria es el producto cartesiano, y no es llevada a cabo ninguna selección en el equijoin, se prefiere ejecutar las selecciones y las proyecciones en primer lugar, dejando el resultado en una relación temporal, como el tamaño de la relación operando influye grandemente en el tiempo que toma ejecutar un producto cartesiano completo.

OPTIMIZACIÓN DE SOLICITUDES DISTRIBUIDAS

La solicitud resultante de descomposición y localización puede ser ejecutada añadiendo, de manera sistemática, primitivas de comunicación. Sin embargo, la permutación del orden de las operaciones dentro de la solicitud puede proporcionar muchas estrategias equivalentes; encontrar un ordenamiento de operaciones óptimo para una solicitud dada es el objetivo del optimizador. Como que la selección de un ordenamiento óptimo para una solicitud es computacionalmente intratable, el objetivo real del optimizador es encontrar una estrategia cercana a la óptima y en general, evitar malas estrategias. La salida del optimizador es una solicitud algebraica sobre fragmentos y operaciones de comunicación para soportar la ejecución de la solicitud sobre los sitios.

La selección de la estrategia optimal generalmente requiere la predicción de los costos de ejecución de los ordenamientos candidatos, los costos de ejecución son expresados como una combinación de E/S, CPU, y costos de comunicación. Es usual ignorar el costo de procesamiento local (E/S y CPU) asumiendo que el costo de comunicación es dominante. La información requerida por el optimizador para estimar los costos de ejecución son estadísticas sobre los fragmentos y fórmulas

para estimar cardinalidades del resultado de operaciones relacionales. El énfasis fundamental se hace en el ordenamiento de los acoples y semiacoples.

Modelo de costo:

El costo de una estrategia de ejecución distribuida puede ser expresado respecto al costo total o al tiempo de respuesta. El costo total es la suma de todos los componentes de costo mientras que el tiempo de respuesta es el tiempo absoluto entre el inicio de la solicitud y su terminación.

$$\text{CostoTotal}=C_{CPU}* \#\text{insts} + C_{E/S} *\# \text{I/Os}+C_{MSG} * \#\text{msgs} + C_{TR} * \#\text{ bytes}.$$

Las dos primeras componentes miden el tiempo de procesamiento local, donde C_{CPU} es el costo de una instrucción y $C_{E/S}$ es el costo de una entrada/salida a disco. El costo de comunicación se refiere a las dos últimas componentes, donde C_{MSG} es el costo fijo de iniciar y recibir un mensaje mientras que C_{TR} es el costo de transmitir una unidad de datos de un sitio a otro (# bytes es la suma de los tamaños de los mensajes) lo que se asume usualmente como constante. El costo de transferir # bytes de datos de un sitio a otro es:

$$cc(\#\text{ bytes}) = C_{MSG} + C_{TR} * \#\text{ bytes}.$$

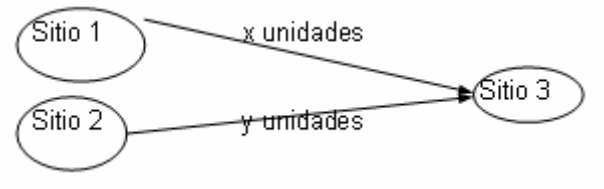
Los costos usualmente se expresan en unidades de tiempo. Los valores relativos de los coeficientes de costo caracterizan el ambiente de BD distribuido y la topología de la red influye significativamente en la proporción entre los componentes del costo: Los SGBDD para redes locales deben considerar los tres costos.

Cuando el tiempo de respuesta de una solicitud es la función objetivo del optimizador, debe considerarse el procesamiento local paralelo y la comunicación paralela.

$$\text{TiempoResp} = C_{CPU} *\text{Seg - \#insts} + C_{E/S} * \text{seg - \#E/Ss} + C_{MSG} * \text{Seg - \#msgs} + C_{TR} * \text{seg - \#bytes}$$

donde seg-# **…* representa el máximo de unidades que deben ser ejecutadas secuencialmente (se ignora cualquier procesamiento o comunicación paralela).

Ejemplo: Observar la diferencia entre costo total y tiempo de respuesta para obtener el resultado de una solicitud en el sitio 3 a partir de datos desde los sitios 1 y 2 (se asumen sólo los costos de comunicación).



Asumir C_{MSG} y C_{TR} en unidades de tiempo. El costo total para transferir x e y el sitio 3 es:

$$\text{Tiempo - total} = 2C_{MSG} + C_{TR} * (x + y)$$

El tiempo de respuesta es aproximadamente:

$$\text{Tiempo - resp} = \text{máx} \{ C_{MSG} + C_{TR} * X, C_{MSG} + C_{TR} * Y\}$$

La minización del tiempo de respuesta se logra incrementando el grado de paralelismo lo cual no implica que se minimice el costo total, la minimización de este supone que se mejore la utilización de los recursos, en la práctica se desea un compromiso entre ambos costos.

Estadísticas sobre la BD:

El principal factor que afecta la estrategia de ejecución es el tamaño de relaciones intermedias que se producen durante la ejecución. Cuando una operación se ubica en un sitio diferente a otro anterior, la relación intermedia debe ser transmitida sobre la red por lo que es necesario estimar el tamaño de estos resultados para minimizar la transferencia de datos. Estos estimados se basan en información estadística sobre las relaciones base y en fórmulas para predecir las cardinalidades de los resultados de operaciones relacionales.

Para una relación R definida sobre los atributos A = $\{A_1,\ldots,A_n\}$ y fragmentada como $\{R_1,\ldots,R_k\}$, los datos estadísticos más usuales, son:

Length (A_i): Longitud del atributo en bytes.

Card ($\Pi_{Ai}(R_j)$): Cantidad de valores distintos para cada atributo A_i en cada fragmento R_j.

Min (A_i) y Máx (A_j): Valores mínimos y máximos respectivamente para atributos definidos sobre un conjunto de valores que pueden ser ordenados.

Card (dom [A_i]): Este valor representa la cantidad de valores únicos en el dominio de cada atributo.

Card (R_j): Cantidad de tuplas de cada fragmento.

Cardinalidad de resultados intermedios:

Se supone una distribución uniforme de los valores de los atributos y una independencia entre los valores de un atributo con los demás. El factor de selectividad de una operación representa la proporción de tuplas que participan en la operación, se denota por: $SF_{op.}$

Selección: Card $\left(\sigma_F(R)\right) = SF_S(F) * card(R)$, donde:

$$SF_S(A = valor) = \frac{1}{card\left(\pi_A(R)\right)}$$

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

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

$$SF_S\left(p(A_i) \wedge p\left(A_j\right)\right) = SF_S\left(p(Ai)\right) * SF_S\left(p\left(A_j\right)\right)$$

$$SF_S\left(p(A_i) \vee p\left(A_j\right)\right) = SF_S\left(p\left(A_i\right)\right) + SF_S\left(p\left(A_j\right)\right) - \left(\mathrm{SF_S}\left(p\left(A_i\right)\right) * SF_s\left(p\left(A_j\right)\right)\right)$$

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

Proyección: Es difícil considerar proyecciones arbitrarias porque usualmente se desconoce la correlación entre atributos proyectados. Hay casos que están bien definidos como:

card (π_A (R)) =card (R) si A es una llave.

Producto cartesiano: card (R X S) = card (R) * card (S)

Acople: No hay una forma general para la cardinalidad del acople sin información adicional. Los SGBD concretos usan algunas cotas o expresiones aproximadas. Hay casos particulares que se pueden estimar como el caso del equijoin donde A es la llave de R y B es una llave extranjera de S referenciando a R, en este caso:

card (R $\bowtie_{A=B}$ S) = card (S).

Este es un valor máximo pues se asume que todo R participa en el acople. Si se mantiene como información estadística del acople, el factor de selectividad de esta operación por algunos pares de relaciones, entonces:

Card (R$\bowtie$S) = SF_J*card (R) * card (S)

Semiacople: Da el por ciento (fracción) de tuplas de R que acoplan con tuplas de S. Una aproximación por su factor de selectividad, es:

$$SF_{SJ}\left(R \propto_A S\right) = \frac{Card\left(\pi_A(S)\right)}{Card\left(dom[A]\right)}$$

Como que esta fórmula sólo depende del atributo A, se denota también por SF_{SJ}(S.A) y representa un factor sobre cualquier otro atributo acoplable.

Card (R$\ltimes_A$(S))=SF_{SJ}(S.A)*card (R)

Unión: Es difícil estimar sin cardinalidad debido a la posible eliminación de duplicados. Sus límites superiores e inferiores son: card (R)+card (S) y max $\{$ card (R), card (S) $\}$ respectivamente.

Diferencia: De forma similar, sus límites superiores e inferiores para card (R-S) son: card (R) respectivamente.

Ordenamiento de los acoples en solicitudes sobre fragmentos:

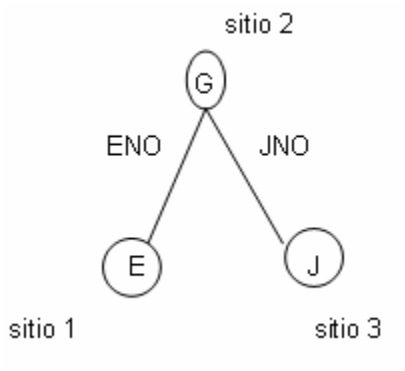
Algunos algoritmos optimizan ordenamiento de acoples directamente sin usar semiacoples. Para su estudio se hacen varias consideraciones. Como que ya se supone que se tiene una solicitud sobre fragmentos, no es necesario distinguir entre un fragmento y otro, sino que se usará el concepto de relación para designar un fragmento almacenado en un sitio particular. Se asumirá que la transferencia es *set-at-a-time*, también se ignora el costo de transferencias para producir el resultado en el sitio final.

El problema más simple es R $\bowtie$ S, donde R y S son relaciones almacenadas en diferentes sitios. La elección obvia es enviar la relación más pequeña al sitio de la más grande; para esto se debe evaluar el tamaño de R y de S.

Cuando hay más de dos relaciones a acoplar, también se debe analizar la transmisión de operandos pequeños. La dificultad se presenta en que las operaciones de acople pueden reducir o incrementar el tamaño de resultados intermedios. Luego, estimar el tamaño de estos resultados es importante pero también difícil. Una solución es estimar los costos de comunicación de todas las estrategias alternativas y elegir la mejor aunque la cantidad de estrategia crece rápidamente con el número de relaciones.

Ejemplo:

Considere la solicitud siguiente expresada en el AR: $J \bowtie_{JNO} E \bowtie_{ENO} G$. Esta solicitud puede ser expresada en 3 lugares diferentes, como por ejemplo:



Cada estrategia se resume a continuación, donde R→ sitio j significa que la relación R se transfiere al sitio j:

1. E→ sitio 2

   sitio 2 computa E' = E ⋈ G

   E' → sitio 3

   sitio 3 computa Res = E' ⋈ J

2. G→ sitio 1

   sitio 1 computa E' = E ⋈ G

   E' → sitio 3

   sitio 3 computa Res = E' ⋈ J

3. G→ sitio 3

   sitio 3 computa G' = G ⋈ J

   G' → sitio 1

   sitio 1 computa Res = G' ⋈ E

4. J→ sitio 2

   sitio 2 computa J' = J ⋈ G

   J' → sitio 1

   sitio 1 computa Res = J' ⋈ E

5. E → sitio 2

   J → sitio 2

   sitio 2 computa Res = E ⋈ J ⋈ G

Para elegir una estrategia deben conocerse o estimarse los tamaños de E, G, J, E⋈G y G⋈J. Una solución heurística puede ser considerar sólo los tamaños de las relaciones operandos, asumiendo como cardinalidad del acople el producto de las cardinalidades de los operandos. Las relaciones se ordenan de manera creciente por sus tamaños, y el orden de ejecución es dado por este ordenamiento y el grafo de acople. Por ejemplo, el orden (E,G,J) puede usar la estrategia 1, mientras que el orden (J,G,E) puede usar la 4.

Algoritmos basados en semi acoples:

El semiacople puede ser usado para decrecer el costo total de solicitudes con acople. La principal limitante de las estrategias descritas con anterioridad es que deben transferirse relaciones completas entre sitios. Un semiacople actúa como un reductor similar a una selección.

El acople de R con S sobre el atributo A, de relaciones almacenadas en los sitios 1 y 2, respectivamente, puede ser computado reemplazando uno o ambos operandos por un semiacople con la otra relación, usando las reglas:

$$R \bowtie_A S \equiv (R \ltimes_A S) \bowtie_A S$$

$$\equiv R \bowtie_A (S \ltimes_A R)$$

$$\equiv (R \ltimes_A S) \bowtie_A (S \ltimes_A R)$$

La elección entre una de estas estrategias requiere estimar sus respectivos costos: El uso de semiacoples es beneficioso si el costo de producirlos y enviarlos a otro sitio es menor que el costo de enviar la relación operando completa y hacer el acople definitivo.

Ejemplo: Comparar el costo de dos alternativas:

1. $R \bowtie_A S$

2. ($R \ltimes_A S$) $\bowtie_A S$ asumiendo size (R) < size (S)

Estrategia usando semiacoples:

1. Π_A (S) $\rightarrow$ sitio 1

2. sitio 1 computa R' = $R \ltimes_A S$

3. R' $\rightarrow$ sitio2

4. sitio 2 computa R' $\bowtie_A S$

Para mayor claridad, se ignora la constante C_{MSG} pues se asume que el término C_{TR} * size (R) es mucho mayor, se comparan las dos alternativas en cuanto a la cantidad de datos transmitidos:

- Costo de estrategia de $R \bowtie_A S$ es el de transferir R al sitio 2.

- Costo de la estrategia de ($R \ltimes_A S$) $\bowtie_A S$ es el de los pasos 1 y 3. Luego esta estrategia es mejor si: size (Π_A (S)) + size ($R \ltimes_A S$) < size (R)

El semiacople es mejor si actúa como buen reductor, esto es, cuando pocas tuplas de R participan en el acople, Si casi todo R participa en el acople es mejor la primera estrategia. El costo de la proyección se puede minimizar codificando su resultado en arreglos de bits. Como que ambas estrategias pueden ser lo mejor, se pueden complementar. Comparado con el acople, el semiacople produce más operaciones pero posiblemente sobre operandos más pequeños.

Algoritmos de optimizacion de solicitud distribuida

Los algoritmos usados por los sistemas INGRES, R* , SDD-1 y otros, se estudian como paradigmas de la fase de optimización, a continuación se presenta uno de ellos: ALGORITMO R*. El algoritmo de optimización de solicitud distribuida de R* "optimiza" en tiempo de compilación con una búsqueda exhaustiva de estrategias alternativas para elegir una con costo mínimo. Aunque la numeración de estas

estrategias es costosa, su demora se compensa si la solicitud se usa con frecuencia posteriormente. El algoritmo de procesamiento de solicitudes de R* no soporta fragmentación ni réplicas, trata solo con relaciones como unidades básicas.

La compilación de la solicitud es una tarea distribuida en R* coordinada por un sitio maestro donde se inició la solicitud. El optimizador del sitio maestro toma todas las decisiones intersitios, mientras que los demás sitios que almacenan datos relacionados con la solicitud toman las demás decisiones locales tales como el ordenamiento de los acoples en el sitio y la generación de planes de accesos locales para la solicitud.
La función objetivo del sistema R* es optimizar costo total incluyendo procesamiento local y costos de comunicación. La entrada al algoritmo de optimización del sistema R* es una solicitud localizada expresada en AR, la ubicación de las relaciones y sus estadísticas.

Algoritmo:
Entrada: QT: Árbol de la solicitud
Salida: strat: estrategia de costo mínimo
begin
for cada relacion $R_i \in$ QT do
begin
for cada camino de acceso AP_{ij} a R_i do
determinar costo (AP_{ij})
end -for
best- $AP_i = AP_{ij}$ con costo mínimo
end
for cada orden ($R_{i1}, R_{i2}, \ldots, R_{in}$) con i=1, …, n! do
begin
construir estrategia $(\ldots((\text{best } AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie \ldots \bowtie R_{in})$
determinar costo de la estrategia
end - for

```
strat: = estrategia con costo mínimo
for cada sitio k almacenando una relación en QT do
begin
        LS_K= estrategia local (estrategia, K)
        enviar (LS_K, sitio k)
    end- for
end
```

El optimizador debe seleccionar el ordenamiento de los acoples y camino de acceso por cada fragmento (índice cluster, búsqueda, secuencial, etc.) Estas decisiones se basan en estadísticas y fórmulas usadas para estimar el tamaño de resultados intermedios e información de los caminos de acceso. El optimizador también debe seleccionar los sitios de los acoples y el método para transferir datos entre sitios.

Para acoplar dos relaciones hay 3 sitio candidatos, el sitio de la primera relación, el sitio de la segunda relación, o un tercer sitio (puede ser el sitio de una tercera relación con la que se debe seguir acoplando). En R* se soportan dos métodos de transferencia de datos intersitios:

1. Transferencia completa: La relación completa es enviada al sitio del acople y almacenada en una relación temporal antes de ser acoplada. Si el algoritmo de acople es de mezcla, la relación no necesita ser almacenada y el sitio del acople puede procesar las tuplas a medida que llegan.
2. Búsqueda iterativa: la relación externa es explorada secuencialmente y se envía el valor de acople de cada tupla al sitio de la relación interna, el que selecciona las tuplas internas que acoplan con el valor recibido y envía las tuplas relacionadas al sitio de la relación externa. Este método es equivalente al semiacople de la relación interna con cada tupla de la externa.

La comparación entre estos dos métodos es obvia. El primero genera una gran transferencia de datos pero con menos mensajes. Esto es ventajoso cuando la relación es pequeña. Si la relación es grande y el acople tiene buena selectividad

(sólo pocas tuplas acoplan) las tuplas de interés deben ser enviadas según se necesita. R* no considera todas las posibles combinaciones de métodos de acople con métodos de transferencia pues algunos no valen la pena. Por ejemplo, pudiera ser inútil transferir la relación externa usando el segundo método en un algoritmo de acoples de lazo anidado.

Dado el acople de una relación externa R con una relación interna S sobre un atributo A, hay 4 estrategias de acople, que a continuación se resumen. LC denota costo de procesamiento local (tiempo de CPU y de E/S) y CC denota costo de comunicación. Se ignora el tiempo de producir el resultado. Se denota por “s” el número promedio de tuplas de S que acoplan con una tupla de R:

$$S = \text{card}\,(S \ltimes_A R) = \text{card}\,(R)$$

Estrategia 1: Transferir la relación externa completa al sitio de la relación interna. En este caso las tuplas externas pueden ser acopladas con S en la medida que lleguen, luego:

costo - total = LC (recuperar card (R) tuplas desde R)
+ CC (size (R))
+ LC (recuperar s tuplas desde S) *card (R)

Estrategia 2: Transferir la relación interna completa al sitio de la relación externa. En este caso las tuplas internas no pueden ser acopladas según llegan y necesitan ser almacenadas en una relación temporal T, luego:

costo - total = LC (recuperar card (S) tuplas desde S)
+ CC (size (S))
+ LC (almacenar card (S) tuplas en T)
+ LC (recuperar card (R) tuplas de R)
+LC (recuperar s tuplas de T)*card (R)

Estrategia 3: Buscar tuplas de la relación interna según se necesite por cada tupla de la relación externa. En este caso, para cada tupla en R se envía el valor de su atributo de acople al sitio de S. Entonces las s tuplas de S que acoplan con dicho valor son recuperadas y enviadas al sitio R para ser acopladas a la medida que lleguen, luego:

costo - total = LC (recuperar card (R) tuplas desde R)

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

+LC (recuperar s tuplas desde s) * card (R)

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

Estrategia 4: Mover ambas relaciones al tercer sitio y computar aquí el acople. En este caso la relación interna es primero movida al tercer sitio y almacenada en una relación temporal T, entonces la relación externa es movida al tercer sitio y sus tuplas son acopladas con T en la medida que llegan, luego:

costo - total = LC (recuperar card (S) tuplas desde S)

+ CC (size (S))

+LC (almacenar card (S) tuplas en T)

+ LC (recuperar card (R) tuplas desde R)

+ CC (size (R))

+ LC (recuperar s tuplas desde T) * card (R)

BIBLIOGRAFÍA

Aho, A. V.; J. E. Hopcroft and J.D. Ullman (1983): Data structures and Algorithms. Addison-Wesley, Reading, Massachussets.

Bernstein, Ph. *et al*. (1987): Concurrency Control and Recovery in Database Systems, Adisson-Wesley Publishing Company.

Ceri, S. and G. Pelagatti (1984): Distributed Databases: Principles and Systems, Mc Graw-Hill.

Korth, H. F. and A. Silberschatz (1991): Fundamentos de Bases de Datos, Mc Graw-Hill Interamericana de España, S. A.

Özsu, M.T. and P. Valduriez (1991): Principles of distributed Database Systems, First Edition, Prentice Hall, Englewood Cliffs, New Jersey.

Rodríguez, A.; L. M. González; C. E. García y M. T. Castro (2007): Bases de Datos en Ambiente Distribuido, Editorial Feijóo, Universidad Central “Marta Abreu” de Las Villas.

Ullman, J. D. (1989): Database and Knowledge-Base System; Computer Science Press.