

5.2 Control de concurrencia distribuida

Como se señaló anteriormente, un algoritmo de control de concurrencia impone un aislamiento particular. En este capítulo, nos centramos principalmente en la serialización entre transacciones concurrentes. La teoría de la serialización se extiende directamente a la distribución. bases de datos. El historial de ejecución de transacciones en cada sitio se denomina historial local. Si la base de datos no se replica y cada historial local es serializable, su unión (llamado historial global) también es serializable siempre que existan órdenes de serialización locales son idénticos

Ejemplo 5.1 Damos un ejemplo muy simple para demostrar el punto. Considere dos cuentas bancarias, x (almacenadas en el Sitio 1) e y (almacenadas en el Sitio 2), y las siguientes dos transacciones donde T1 transfiere \$100 de x a y, mientras que T2 simplemente lee el equilibrio de x e y:

T1: Leer(x)	T2: Leer(x)
$x \leftarrow x - 100$	Listo)
Escribe(x)	Comprometerse
Listo)	
$y \leftarrow y + 100$	
Escribe(y)	
Comprometerse	

Obviamente, ambas transacciones deben ejecutarse en ambos sitios. Considere la siguientes dos historias que pueden generarse localmente en los dos sitios (Hola es el EI historial en el Sitio i y Rj y Wj son operaciones de Lectura y Escritura , respectivamente, en transacción Tj):

$$H_1 = \{R1(x), W1(x), R2(x)\}$$

$$H_2 = \{R1(y), W1(y), R2(y)\}$$

Ambas historias son serializables; de hecho, son seriales. Por lo tanto, cada una representa un orden de ejecución correcto. Además, el orden de serialización para ambos Son los mismos $T1 \rightarrow T2$. Por lo tanto, el historial global obtenido también es serializable. con el orden de serialización $T1 \rightarrow T2$.

Sin embargo, si las historias generadas en los dos sitios son las siguientes, hay una problema:

$$H_1 = \{R1(x), W1(x), R2(x)\}$$

$$H_2 = \{R2(y), R1(y), W1(y)\}$$

Aunque cada historia local todavía se puede serializar, los órdenes de serialización son diferente: H_1 serializa T1 antes de T2 ($T1 \rightarrow T2$) mientras H_2 serializa T2 antes de T1 ($T2 \rightarrow T1$). Por lo tanto, no puede haber un historial global serializable.

Los protocolos de control de concurrencia se encargan del aislamiento. Un protocolo busca garantizar un nivel de aislamiento específico, como la serialización, el aislamiento de instantáneas o la confirmación de lectura. El control de concurrencia tiene diferentes aspectos o dimensiones. El primero es, obviamente, el nivel de aislamiento que busca el algoritmo. El segundo aspecto es si un protocolo impide que se rompa el aislamiento (pesimista) o si lo permite y luego cancela una de las transacciones conflictivas para preservar el nivel de aislamiento (optimista).

La tercera dimensión se refiere a cómo se serializan las transacciones. Pueden serializarse según el orden de los accesos conflictivos o según un orden predefinido, denominado orden de marca de tiempo. El primer caso corresponde a los algoritmos de bloqueo, donde las transacciones se serializan según el orden en que intentan adquirir bloqueos conflictivos. El segundo caso corresponde a los algoritmos que ordenan las transacciones según una marca de tiempo. Esta marca de tiempo puede asignarse al inicio de la transacción (marca de tiempo de inicio) si son pesimistas o justo antes de confirmar la transacción (marca de tiempo de confirmación) si son optimistas. La cuarta dimensión que consideramos es cómo se mantienen las actualizaciones. Una posibilidad es mantener una única versión de los datos (lo cual es posible en algoritmos pesimistas). Otra posibilidad es mantener múltiples versiones de los datos. Esto último es necesario para los algoritmos optimistas, pero algunos algoritmos pesimistas también lo utilizan para fines de recuperación (básicamente, mantienen dos versiones: la última confirmada y la actual sin confirmar). Analizaremos la replicación en el siguiente capítulo.

Se han explorado la mayoría de las combinaciones de estas múltiples dimensiones. En el resto de esta sección, nos centraremos en las técnicas fundamentales para algoritmos pesimistas: bloqueo (Sección 5.2.1) y ordenación de marcas de tiempo (Sección 5.2.2), así como los optimistas (Sección 5.2.4). Comprender estas técnicas también ayudará al lector a avanzar hacia algoritmos más complejos de este tipo.

5.2.1 Algoritmos basados en bloqueo

Los algoritmos de control de concurrencia basados en bloqueos previenen la violación del aislamiento manteniendo un bloqueo para cada unidad de bloqueo y requiriendo que cada operación obtenga un bloqueo en el elemento de datos antes de acceder a él, ya sea en modo de lectura (compartido) o de escritura (exclusivo). La solicitud de acceso de la operación se decide en función de la compatibilidad de los modos de bloqueo: un bloqueo de lectura es compatible con otro bloqueo de lectura, un bloqueo de escritura no es compatible con un bloqueo de lectura o escritura. El sistema gestiona los bloqueos mediante el algoritmo de bloqueo de dos fases. La decisión fundamental en los algoritmos de control de concurrencia distribuidos basados en bloqueos es dónde y cómo se mantienen los bloqueos (generalmente denominados tabla de bloqueos). Las siguientes secciones proporcionan algoritmos que toman diferentes decisiones al respecto.

5.2.1.1 2PL centralizado

El algoritmo 2PL puede extenderse fácilmente al entorno de SGBD distribuido delegando la responsabilidad de la gestión de bloqueos a un solo sitio. Esto significa que solo uno de los sitios tiene un administrador de bloqueos; los administradores de transacciones de los demás sitios se comunican con él para obtener bloqueos. Este enfoque también se conoce como algoritmo 2PL de sitio principal .

La comunicación entre los sitios participantes al ejecutar una transacción según un algoritmo 2PL centralizado (C2PL) se muestra en la Fig. 5.3, donde se indica el orden de los mensajes. Esta comunicación se lleva a cabo entre el TM coordinador , el gestor de bloqueos del sitio central y los procesadores de datos (PD) de los demás sitios participantes. Los sitios participantes son aquellos que almacenan los datos sobre los que se realizará la operación.

El algoritmo de gestión de transacciones 2PL centralizado (C2PL-TM) que incorpora estos cambios se proporciona en un nivel muy alto en el algoritmo 5.1, mientras que el algoritmo de gestión de bloqueos 2PL centralizado (C2PL-LM) se muestra en el algoritmo 5.2.

En el algoritmo 5.3 se presenta un algoritmo de procesador de datos (PD) muy simplificado , que experimentará cambios importantes cuando analicemos cuestiones de confiabilidad en la sección 5.4.

Estos algoritmos utilizan una quintupla para la operación que realizan: Op: T ype = {BT, R, W, A, C}, arg: Elemento de datos, val: Valor, tid: Identificador de la transacción, res: Resultado. Para una operación o: Op, oT ype = {BT, R, W, A, C} especifica su tipo, donde BT = Begin_transaction, R = Read, W = Write, A = Abort y C = Commit. arg es el elemento de datos al que accede la operación (lee o escribe; para otras operaciones, este campo es nulo), val es el valor leído o por escribir del elemento de datos arg.

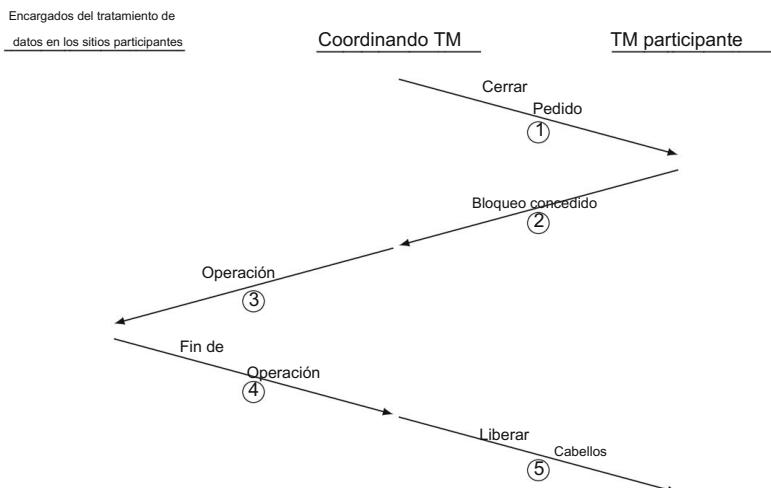


Fig. 5.3 Estructura de comunicación del 2PL centralizado

Algoritmo 5.1: Gestor de transacciones 2PL centralizado (C2PL-TM)

Entrada: msg: un mensaje



Algoritmo 5.2: Gestor de bloqueos 2PL centralizado (C2PL-LM)

```

Entrada: op : Op
begin
    switch op.Type do case
        R o W do encuentra
            la unidad de bloqueo lu tal que op.arg    lu si lu está
            desbloqueado o el modo de bloqueo de lu es compatible con op.T type entonces establece
                el bloqueo en lu en el modo apropiado en nombre de la transacción op.t.id envía
                "Bloqueo concedido" al TM coordinador de la transacción de lo contrario

            coloca op en una cola para lu fin si

        fin de caso
        caso C o A hacer
            foreach unidad de bloqueo lu retenida por la transacción
                hacer liberar el bloqueo en lu retenido por la
                transacción si hay operaciones esperando en la cola para lu
                    entonces encontrar la primera operación O
                    en la cola establecer un bloqueo en lu
                    en nombre de O enviar "Bloqueo concedido" al TM coordinador de la transacción
                    Ot id fin si
            fin de foreach
            envía "Bloqueos liberados" al TM coordinador del caso final de la
            transacción
    interruptor final
fin

```

(para otras operaciones es nulo), tid es la transacción a la que pertenece esta operación (estrictamente hablando, este es el identificador de la transacción), y res indica el código de finalización de las operaciones solicitadas a DP, lo cual es importante para los algoritmos de confiabilidad.

El algoritmo del gestor de transacciones (C2PL-TM) se escribe como un proceso que se ejecuta indefinidamente y espera la llegada de un mensaje, ya sea de una aplicación (con una operación de transacción), de un gestor de bloqueos o de un procesador de datos. Los algoritmos del gestor de bloqueos (C2PL-LM) y del procesador de datos (PD) se escriben como procedimientos que se invocan cuando es necesario. Dado que los algoritmos se presentan con un alto nivel de abstracción, esto no supone un problema importante, pero las implementaciones reales pueden, naturalmente, ser bastante diferentes.

Una crítica común a los algoritmos C2PL es la rápida formación de cuellos de botella en torno al sitio central. La comunicación entre los sitios que cooperan al ejecutar una transacción según un algoritmo 2PL centralizado (C2PL) se muestra en la Figura 5.3, donde se indica el orden de los mensajes. Esta comunicación se realiza entre el TM coordinador , el gestor de bloqueos del sitio central y los procesadores de datos (PD) de los demás sitios participantes. Los sitios participantes son aquellos que almacenan los datos sobre los que se realizará la operación. Además, el sistema puede ser menos fiable, ya que un fallo o la inaccesibilidad del sitio central provocaría fallos graves.

Algoritmo 5.3: Procesador de Datos (PD)

```

Entrada: op : Op begin
switch
    op.Type hacer caso BT
        hacer algo de
            | contabilidad fin caso caso R
            hacer

            | op.res ← LEER(op.arg) ; op.res ←
            | "Lectura realizada" fin caso caso
            W hacer
                {verifique el tipo de operación} {los
                detalles se discutirán en la Sección 5.4}
                {operación de lectura de base de datos}

        ESCRIBIR(op.arg, op.val) op.res ←
            | "Escritura realizada" fin caso
                {ESCRITURA de base de datos de val en el elemento de datos arg}

        caso C hacer
            | COMMIT; op.res
            | ← "Commit hecho" fin caso caso A
            hacer
                {ejecutar COMMIT}

        ABORT;
            | op.res ←
            | "Abort hecho" fin caso
                {ejecutar ABORT}

    interruptor final
    retorno final de la
operación

```

5.2.1.2 2PL distribuido

El protocolo 2PL distribuido (D2PL) requiere la disponibilidad de administradores de bloqueo en cada sitio. La comunicación entre sitios cooperantes que ejecutan una transacción según el protocolo 2PL distribuido se muestra en la Figura 5.4.

El algoritmo distribuido de gestión de transacciones 2PL es similar al de C2PL-TM, con dos modificaciones importantes. Los mensajes que se envían al gestor de bloqueos del sitio central en C2PL-TM se envían a los gestores de bloqueos de todos los sitios participantes en D2PL-TM. La segunda diferencia radica en que las operaciones no son transferidas a los procesadores de datos por el gestor de transacciones coordinador, sino por los gestores de bloqueos participantes. Esto significa que el gestor de transacciones coordinador no espera un mensaje de "solicitud de bloqueo concedida". Otro aspecto importante de la figura 5.4 es el siguiente. Los procesadores de datos participantes envían los mensajes de "fin de operación" al TM coordinador. La alternativa es que cada DP los envíe a su propio gestor de bloqueos, quien puede liberar los bloqueos e informar al TM coordinador. Hemos optado por describir el primero, ya que utiliza un algoritmo de LM idéntico al gestor de bloqueos 2PL estricto que ya hemos analizado y simplifica la explicación de los protocolos de confirmación (véase la sección 5.4). Debido a estas similitudes, no presentamos...

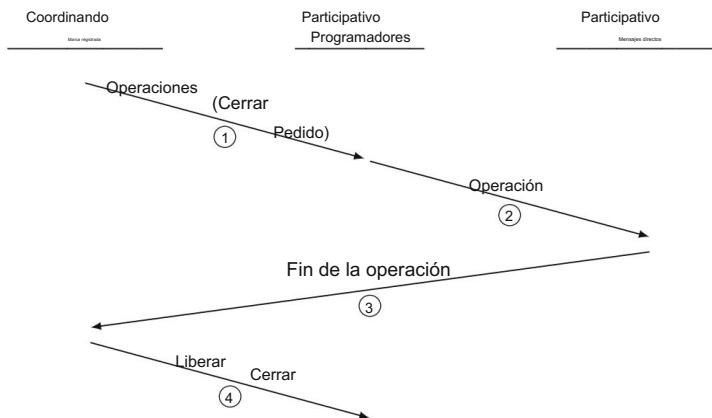


Fig. 5.4 Estructura de comunicación de 2PL distribuido

Algoritmos TM y LM distribuidos. Los algoritmos 2PL distribuidos se han utilizado en R* y NonStop SQL.

5.2.1.3 Gestión distribuida de bloqueos

Los algoritmos de control de concurrencia basados en bloqueos pueden causar interbloqueos; en el caso de los DBMS distribuidos, estos podrían ser interbloqueos distribuidos (o globales) debido a transacciones que se ejecutan en diferentes sitios esperando entre sí. La detección y resolución de interbloqueos es el enfoque más popular para la gestión de interbloqueos en el entorno distribuido. El grafo de espera (WFG) puede ser útil para detectar interbloqueos; este es un grafo dirigido cuyos vértices son transacciones activas con una arista de T_i a T_j si una operación en T_i está esperando para acceder a un elemento de datos que actualmente está bloqueado en un modo incompatible por una operación en T_j . Sin embargo, la formación del WFG es más complicada en un entorno distribuido debido a la ejecución distribuida de transacciones. Por lo tanto, no es suficiente que cada sitio forme un grafo de espera local (LWFG) y lo revise; también es necesario formar un grafo de espera global (GWFG), que es la unión de todos los LWFG, y revisarlo para ciclos.

Ejemplo 5.2. Considere cuatro transacciones T_1 , T_2 , T_3 y T_4 con la siguiente relación de espera: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. Si T_1 y T_2 se ejecutan en el sitio 1, mientras que T_3 y T_4 se ejecutan en el sitio 2, las LWFG de ambos sitios se muestran en la Fig. 5.5a. Tenga en cuenta que no es posible detectar un interbloqueo simplemente examinando las dos LWFG de forma independiente, ya que el interbloqueo es global. Sin embargo, el interbloqueo se puede detectar fácilmente examinando la GWFG, donde la espera entre sitios se muestra con líneas discontinuas (Fig. 5.5b).



Figura 5.5 Diferencia entre LWFG y GWFG

Los distintos algoritmos difieren en la gestión del GWFG. Existen tres métodos fundamentales para detectar interbloqueos distribuidos: centralizado, distribuido y jerárquico. Los analizamos a continuación.

Detección centralizada de bloqueos

En el enfoque de detección centralizada de interbloqueos, se designa un sitio como detector de interbloqueos para todo el sistema. Periódicamente, cada gestor de bloqueos transmite su LWFG al detector de interbloqueos, que a su vez forma el GWFG y busca ciclos en él. Los gestores de bloqueos solo necesitan enviar los cambios en sus grafos (es decir, los bordes recién creados o eliminados) al detector de interbloqueos. La duración de los intervalos para transmitir esta información depende del diseño del sistema: cuanto menor sea el intervalo, menores serán los retrasos debidos a interbloqueos no detectados, pero mayor será la detección de interbloqueos y la sobrecarga de comunicación.

La detección centralizada de interbloqueos es sencilla y sería una opción natural si el algoritmo de control de concurrencia fuera un 2PL centralizado. Sin embargo, también deben considerarse los problemas de vulnerabilidad a fallos y la alta sobrecarga de comunicación.

Detección de bloqueos jerárquicos

Una alternativa a la detección centralizada de interbloqueos es la creación de una jerarquía de detectores de interbloqueos (véase la figura 5.6). Los interbloqueos locales en un único sitio se detectarían en dicho sitio mediante el LWFG. Cada sitio, a su vez, envía su LWFG al detector de interbloqueos del nivel superior. Por lo tanto, los interbloqueos distribuidos que afecten a dos o más sitios se detectarían mediante un detector de interbloqueos del nivel inferior que tenga control sobre estos sitios. Por ejemplo, un interbloqueo en el sitio 1 se detectaría mediante el detector de interbloqueos local (DD) del sitio 1 (denotado DD21, 2 para el nivel 2, 1 para el sitio 1). Sin embargo, si el interbloqueo afecta a los sitios 1 y 2, DD11 lo detecta. Finalmente, si el interbloqueo afecta a los sitios 1 y 4, DD0x lo detecta, donde x es uno de los siguientes: 1, 2, 3 o 4.

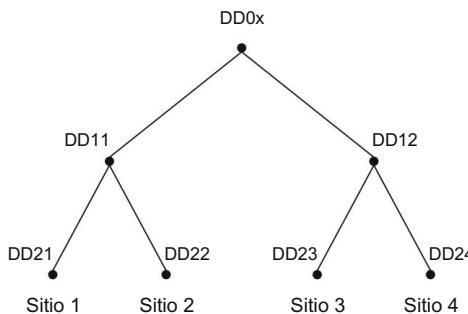


Fig. 5.6 Detección de bloqueo jerárquico

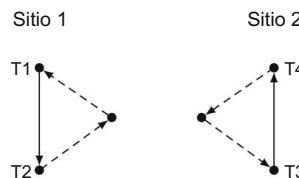


Fig. 5.7 LWFG modificados

El método jerárquico de detección de interbloqueos reduce la dependencia del sitio central, lo que reduce el coste de la comunicación. Sin embargo, su implementación es considerablemente más compleja e implicaría modificaciones significativas en los algoritmos del gestor de bloques y transacciones.

Detección de bloques distribuidos

Los algoritmos distribuidos de detección de interbloqueos delegan la responsabilidad de detectarlos a sitios individuales. Por lo tanto, al igual que en la detección jerárquica de interbloqueos, existen detectores locales de interbloqueos en cada sitio que comunican sus LWFG entre sí (de hecho, solo se transmiten los ciclos de interbloqueo potenciales). Entre los diversos algoritmos distribuidos de detección de interbloqueos, el implementado en System R* es el más conocido y referenciado, y lo describimos a continuación.

El LWFG en cada sitio se forma y se modifica de la siguiente manera:

1. Dado que cada sitio recibe los ciclos de bloqueo potencial de otros sitios, estos Se añaden bordes a los LWFG.
2. Los bordes en el LWFG que muestran que las transacciones locales están esperando transacciones en otros sitios se unen con los bordes en los LWFG que representan que las transacciones remotas están esperando transacciones locales.

Ejemplo 5.3. Considere el ejemplo de la Fig. 5.5. El WFG local de ambos sitios se modifica como se muestra en la Fig. 5.7.

Los detectores de interbloqueo local buscan dos cosas. Si existe un ciclo que no incluye los flancos externos, existe un interbloqueo local que puede gestionarse localmente. Si, por otro lado, existe un ciclo que involucra estos flancos externos, existe un posible interbloqueo distribuido, y esta información del ciclo debe comunicarse a otros detectores de interbloqueo. En el caso del Ejemplo 5.3, ambos sitios detectan la posibilidad de dicho interbloqueo distribuido.

Una pregunta que debe responderse en este punto es a quién transmitir la información. Obviamente, se puede transmitir a todos los detectores de interbloqueo del sistema. A falta de más información, esta es la única alternativa, pero conlleva una alta sobrecarga. Sin embargo, si se sabe si la transacción está adelantada o retrasada en el ciclo de interbloqueo, la información puede transmitirse hacia adelante o hacia atrás a lo largo de los sitios de este ciclo. El sitio receptor modifica entonces su LWFG como se mencionó anteriormente y comprueba si hay interbloqueos. Obviamente, no es necesario transmitir a lo largo del ciclo de interbloqueo tanto hacia adelante como hacia atrás. En el caso del Ejemplo 5.3, el sitio 1 la enviaría al sitio 2 tanto hacia adelante como hacia atrás a lo largo del ciclo de interbloqueo.

Los algoritmos distribuidos de detección de interbloqueos requieren una modificación uniforme de los administradores de bloqueos en cada sitio. Esta uniformidad facilita su implementación. Sin embargo, existe la posibilidad de una transmisión excesiva de mensajes. Esto ocurre, por ejemplo, en el caso del Ejemplo 5.3: el sitio 1 envía su información de posible interbloqueo al sitio 2, y el sitio 2 envía su información al sitio 1. En este caso, los detectores de interbloqueo de ambos sitios detectarán el interbloqueo. Además de causar una transmisión innecesaria de mensajes, existe el problema adicional de que cada sitio puede elegir una víctima diferente para abortar. El algoritmo de Obermarck resuelve el problema utilizando marcas de tiempo de transacción (contador de incremento monótono; véanse más detalles en la siguiente sección), así como la siguiente regla. Sea $T_i \rightarrow \dots \rightarrow T_j$ la ruta que tiene el potencial de causar un interbloqueo distribuido en el WFG local de un sitio .

Un detector de bloqueo local reenvía la información del ciclo solo si la marca de tiempo de T_i es menor que la marca de tiempo de T_j . Esto reduce el número promedio de transmisiones de mensajes a la mitad. En el caso del Ejemplo 5.3, el sitio 1 tiene una ruta $T_1 \rightarrow T_2 \rightarrow T_3$, mientras que el sitio 2 tiene una ruta $T_3 \rightarrow T_4 \rightarrow T_1$. Por lo tanto, suponiendo que los subíndices de cada transacción indican su marca de tiempo, solo el sitio 1 enviará información al sitio 2.

5.2.2 Algoritmos basados en marcas de tiempo

Los algoritmos de control de concurrencia basados en marcas de tiempo seleccionan, a priori, un orden de serialización y ejecutan las transacciones según este. Para establecer este orden, el gestor de transacciones asigna a cada transacción T_i una marca de tiempo única , $ts(T_i)$, al iniciarse.

La asignación de marcas de tiempo en un SGBD distribuido requiere atención, ya que varios sitios asignarán marcas de tiempo, y mantener la unicidad y la monotonía en todo el sistema no es fácil. Un método consiste en utilizar un contador global (para todo el sistema) de crecimiento monótono. Sin embargo, el mantenimiento de los contadores globales es un problema en los sistemas distribuidos. Por lo tanto, es preferible que cada sitio asigne marcas de tiempo de forma autónoma en función de su contador local. Para mantener la unicidad, cada sitio añade su propio identificador al valor del contador. Por lo tanto, la marca de tiempo es una doble tupla con la forma valor del contador local, identificador del sitio. Tenga en cuenta que el identificador del sitio se añade en la posición menos significativa. Por lo tanto, solo sirve para ordenar las marcas de tiempo de dos transacciones a las que se les podría haber asignado el mismo valor del contador local. Si cada sistema puede acceder a su propio reloj del sistema, es posible utilizar los valores del reloj del sistema en lugar de los valores del contador.

Arquitectónicamente (véase la Fig. 5.2), el gestor de transacciones se encarga de asignar una marca de tiempo a cada nueva transacción y adjuntarla a cada operación de la base de datos que pasa al planificador. Este último componente se encarga de registrar las marcas de tiempo de lectura y escritura, así como de realizar la comprobación de serialización.

5.2.2.1 Algoritmo TO básico

En el algoritmo TO básico, la TM coordinadora asigna la marca de tiempo a cada transacción T_i [$ts(T_i)$], determina los sitios donde se almacena cada dato y envía las operaciones pertinentes a estos sitios. Se trata de una implementación sencilla de la regla TO.

Regla TO: Dadas dos operaciones en conflicto, O_{ij} y O_{kl} , pertenecientes, respectivamente, a las transacciones T_i y T_k , O_{ij} se ejecuta antes que O_{kl} si y solo si $ts(T_i) < ts(T_k)$. En este caso, se dice que T_i es la transacción más antigua y T_k , la más reciente .

Un programador que aplica la regla TO verifica cada nueva operación con operaciones conflictivas ya programadas. Si la nueva operación pertenece a una transacción más reciente que todas las conflictivas ya programadas, se acepta; de lo contrario, se rechaza, lo que provoca que la transacción se reinicie con una nueva marca de tiempo.

Para facilitar la comprobación de la regla TO, a cada elemento de datos x se le asignan dos marcas de tiempo: una marca de tiempo de lectura [$rts(x)$], que corresponde a la mayor de las transacciones que han leído x , y una marca de tiempo de escritura [$wts(x)$], que corresponde a la mayor de las transacciones que han escrito (actualizado) x . Ahora basta con comparar la marca de tiempo de una operación con las marcas de tiempo de lectura y escritura del elemento de datos al que se desea acceder para determinar si alguna transacción con una marca de tiempo mayor ya ha accedido al mismo elemento de datos.

El algoritmo básico del gestor de transacciones TO (BTO-TM) se describe en el algoritmo 5.4. Los históricos de cada sitio simplemente aplican la regla TO. El programador

Algoritmo 5.4: Ordenamiento básico de marcas de tiempo (BTO-TM)

Entrada: msg : un mensaje

comenzar

- repetir
 - esperar un mensaje
 - cambiar el tipo de mensaje
 - caso transacción operación hacer sea op {operación del programa de aplicación}
 - la operación
 - interruptor op.Tipo hacer
 - caso BT hacer
 - S ← ; {S: conjunto de sitios donde se ejecuta la transacción}
 - asigna una marca de tiempo a la transacción; llámala ts(T)
 - Caso final DP(op) {llamar a DP con operación}
 - caso R, W hacen
 - buscar el sitio que almacena el elemento de datos solicitado (por ejemplo, Si)
 - BTO-SCSi(op, ts(T)) ; {enviar op y ts a SC en Si}
 - S ← S Si {crear lista de sitios donde se ejecutan transacciones}
 - caso final
 - caso A, C hacen {enviar op a los DP que ejecutan la transacción}
 - | DPS (op)
 - caso final
 - interruptor final
- caso final
- caso SC respuesta hacer
 - op.T type ← A; BTO-
 - {La operación debe haber sido rechazada por un SC}
 - SCS (op, -) ; reiniciar la transacción con una nueva marca de tiempo {preguntar a otros SC participantes}
- caso final
- caso DP respuesta hacer {mensaje de operación completada}
 - cambiar tipo de operación de transacción hacer
 - sea op la operación
 - caso R devuelve op.val a la aplicación
 - Caso W informar a la solicitud de la finalización del escrito
 - caso C hacer
 - Si se ha recibido el mensaje de confirmación de todos los participantes, entonces
 - | Informar a la aplicación sobre la finalización exitosa de la transacción
 - demás {esperar hasta que lleguen los mensajes de confirmación de todos}
 - | registrar la llegada del mensaje de confirmación
 - terminar si
 - caso final
 - caso A hacer
 - Informar a la aplicación de la finalización del aborto
 - Caso final BTO-
 - SC(op) {Es necesario reiniciar los ts de lectura y escritura}
 - interruptor final
- caso final
- interruptor final
- hasta siempre

El algoritmo se describe en el Algoritmo 5.5. El administrador de datos sigue siendo el del Algoritmo 5.3. Las mismas estructuras de datos y suposiciones que utilizamos para los algoritmos 2PL centralizados se aplican también a estos algoritmos.

Cuando un programador rechaza una operación, el gestor de transacciones reinicia la transacción correspondiente con una nueva marca de tiempo. Esto garantiza que la transacción pueda ejecutarse en su siguiente intento. Dado que las transacciones nunca esperan mientras tienen derechos de acceso a los datos, el algoritmo TO básico nunca causa interbloqueos. Sin embargo, la desventaja de no sufrir interbloqueos es la posibilidad de reiniciar una transacción varias veces. Existe una alternativa al algoritmo TO básico que reduce el número de reinicios, que se describe en la siguiente sección.

Otro detalle a considerar se relaciona con la comunicación entre el programador y el procesador de datos. Cuando una operación aceptada se transfiere al procesador de datos, el programador debe abstenerse de enviar otra operación conflictiva, pero aceptable, hasta que la primera sea procesada y confirmada. Esto es necesario para garantizar que el procesador de datos ejecute las operaciones en el orden en que el programador las transfiere. De lo contrario, las marcas de tiempo de lectura y escritura del dato accedido no serían precisas.

Algoritmo 5.5: Programador básico de ordenación de marcas de tiempo (BTO-SC)

Entrada: op : Op; ts(T) : Marca de tiempo

```

comenzar recuperar rts(op.arg) y wts(arg) guardar
    rts(op.arg) y wts(arg) ; cambiar op.arg                                {podría ser necesario si se aborta}
    hacer caso R hacer si
        ts(T) >
            wts(op.arg) entonces DP(op);                                     {la operación se puede ejecutar; enviarla a DP}
            rts(op.arg)
            ← ts(T) de lo contrario

            enviar mensaje "Rechazar transacción" a la TM coordinadora fin si

        caso final

    caso W hacer
        si ts(T) > rts(op.arg) y ts(T) > wts(op.arg) entonces DP(op);
            rts(op.arg)                                              {la operación se puede ejecutar; enviarla a DP}
            ← ts(T) wts(op.arg) ← ts(T)
            de lo contrario enviar mensaje

            "Rechazar transacción" al TM coordinador fin si

        fin de caso

    caso A hacer
        para todo op.arg al que se ha accedido mediante la transacción hacer
            | restablece rts(op.arg) y wts(op.arg) a sus valores iniciales. fin para todo

        caso final

    interruptor final
fin

```

Ejemplo 5.4 Supongamos que el planificador TO recibe primero $Wi(x)$ y luego $Wj(x)$, donde $ts(Ti) < ts(Tj)$. El planificador aceptaría ambas operaciones y las pasaría al procesador de datos. El resultado de estas dos operaciones es que $wts(x) = ts(Tj)$, y entonces esperamos que el efecto de $Wj(x)$ se represente en la base de datos. Sin embargo, si el procesador de datos no las ejecuta en ese orden, los efectos en la base de datos serán erróneos.

El programador puede imponer el orden manteniendo una cola para cada elemento de datos, lo que permite retrasar la transferencia de la operación aceptada hasta que se reciba una confirmación del procesador de datos sobre la operación anterior realizada en el mismo elemento. Este detalle no se muestra en el algoritmo 5.5.

Esta complicación no surge en algoritmos basados en 2PL, ya que el administrador de bloqueos ordena las operaciones liberando los bloqueos solo después de su ejecución. En cierto sentido, la cola que mantiene el programador TO puede considerarse un bloqueo. Sin embargo, esto no implica que el historial generado por un programador TO y uno 2PL sea siempre equivalente. Hay algunos históricos generados por un programador TO que no serían admisibles por un historial 2PL.

Recuerde que, en el caso de algoritmos 2PL estrictos, la liberación de bloqueos se retrasa aún más, hasta la confirmación o cancelación de una transacción. Es posible desarrollar un algoritmo TO estricto utilizando un esquema similar. Por ejemplo, si $Wi(x)$ se acepta y se libera al procesador de datos, el programador retrasa todas las operaciones $Rj(x)$ y $Wj(x)$ (para todos los Tj) hasta que Ti finalice (confirme o cancele).

5.2.2.2 Algoritmo TO conservador

En la sección anterior, indicamos que el algoritmo TO básico nunca hace que las operaciones esperen, sino que las reinicia. También señalamos que, si bien esto representa una ventaja debido a la ausencia de interbloqueos, también es una desventaja, ya que numerosos reinicios tendrían consecuencias negativas para el rendimiento. Los algoritmos TO conservadores intentan reducir esta sobrecarga del sistema reduciendo el número de reinicios de transacciones.

Primero, presentemos una técnica común para reducir la probabilidad de reinicios. Recuerde que un programador TO reinicia una transacción si ya se ha programado o ejecutado una transacción conflictiva más reciente. Tenga en cuenta que estas incidencias aumentan significativamente si, por ejemplo, un sitio está relativamente inactivo en comparación con los demás y no emite transacciones durante un período prolongado. En este caso, su contador de marca de tiempo indica un valor considerablemente menor que los contadores de los otros sitios. Si el TM de este sitio recibe una transacción, las operaciones enviadas a los históricos de los otros sitios serán rechazadas con casi total seguridad, lo que provocará el reinicio de la transacción. Además, la misma transacción se reiniciará repetidamente hasta que el valor del contador de marca de tiempo en su sitio de origen alcance la paridad con los contadores de los otros sitios.

El escenario anterior indica que es útil mantener sincronizados los contadores de cada sitio. Sin embargo, la sincronización total no solo es costosa (ya que requiere el intercambio de mensajes cada vez que cambia un contador), sino también innecesaria. En su lugar, cada gestor de transacciones puede enviar sus operaciones remotas, en lugar de históricas, a los gestores de transacciones de los demás sitios. Los gestores de transacciones receptores pueden entonces comparar sus propios valores de contador con los de la operación entrante. Cualquier gestor cuyo valor de contador sea menor que el entrante ajusta su propio contador a uno mayor que el entrante. Esto garantiza que ningún contador del sistema se descontrolle ni se retrase significativamente. Por supuesto, si se utilizan relojes del sistema en lugar de contadores, esta sincronización aproximada puede lograrse automáticamente siempre que los relojes estén sincronizados con un protocolo como el Protocolo de Tiempo de Red (NTP).

Los algoritmos TO conservadores ejecutan cada operación de manera diferente al TO básico. El algoritmo TO básico intenta ejecutar una operación en cuanto se acepta; por lo tanto, es "agresivo" o "progresivo". Los algoritmos conservadores, en cambio, retrasan cada operación hasta garantizar que ninguna operación con una marca de tiempo menor pueda llegar a ese planificador. Si se puede garantizar esta condición, el planificador nunca rechazará una operación. Sin embargo, este retraso puede generar interbloqueos.

La técnica básica que se utiliza en el TO conservador se basa en la siguiente idea: las operaciones de cada transacción se almacenan en un buffer hasta que se pueda establecer un orden tal que no sean posibles los rechazos, y se ejecutan en ese orden.

Consideraremos una posible implementación del algoritmo TO conservador.

Supongamos que cada programador mantiene una cola para cada gestor de transacciones del sistema. El programador del sitio s almacena todas las operaciones que recibe del gestor de transacciones del sitio t en la cola Qt. El programador del sitio s tiene una cola de este tipo para cada sitio t. Cuando se recibe una operación de un gestor de transacciones, se coloca en la cola correspondiente en orden creciente de fecha y hora. Los históricos de cada sitio ejecutan las operaciones de estas colas en orden creciente de fecha y hora.

Este esquema reducirá el número de reinicios, pero no garantiza su eliminación completa. Considere el caso donde en el sitio s la cola para el sitio t (Qt s) está vacía. El planificador en el sitio s seleccionará una operación [digamos, R(x)] con la marca de tiempo más pequeña y la pasará al procesador de datos. Sin embargo, el sitio t podría haber enviado a s una operación [digamos, W(x)] con una marca de tiempo más pequeña, que aún podría estar en tránsito en la red. Cuando esta operación llega al sitio s, será rechazada, ya que infringe la regla TO: desea acceder a un elemento de datos al que está accediendo actualmente (en un modo incompatible) otra operación con una marca de tiempo más alta.

Es posible diseñar un algoritmo TO extremadamente conservador al exigir que el programador seleccione una operación para enviar al procesador de datos solo si hay al menos una operación en cada cola. Esto garantiza que cada operación que el programador reciba en el futuro tendrá marcas de tiempo mayores o iguales a las que se encuentran actualmente en las colas. Por supuesto, si un gestor de transacciones no tiene ninguna transacción que procesar, debe enviar mensajes ficticios periódicamente a todos los programadores del sistema para informarles que las operaciones que enviará en el futuro tendrán marcas de tiempo mayores que las del mensaje ficticio.

El lector atento se dará cuenta de que el programador de ordenamiento de marcas de tiempo, extremadamente conservador, ejecuta transacciones en serie en cada sitio. Esto es muy restrictivo. Un método empleado para superar esta restricción consiste en agrupar las transacciones en clases. Las clases de transacción se definen con respecto a sus conjuntos de lectura y escritura. Por lo tanto, basta con determinar la clase a la que pertenece una transacción comparando los conjuntos de lectura y escritura de la transacción, respectivamente, con los de cada clase. Así, el algoritmo TO conservador puede modificarse para que, en lugar de requerir la existencia, en cada sitio, de una cola para cada gestor de transacciones, solo sea necesaria una cola para cada clase de transacción. Alternativamente, se podría marcar cada cola con la clase a la que pertenece. Con cualquiera de estas modificaciones, se modifican las condiciones para enviar una operación al procesador de datos. Ya no es necesario esperar a que haya al menos una operación en cada cola; basta con esperar a que haya al menos una operación en cada clase a la que pertenece la transacción. Esta y otras condiciones menos rigurosas que reducen el tiempo de espera pueden definirse y son suficientes. Una variante de este método se utiliza en el sistema prototipo SDD-1.

5.2.3 Control de concurrencia multiversión

Los enfoques que analizamos anteriormente abordan fundamentalmente las actualizaciones *in situ*: cuando se actualiza el valor de un elemento de datos, su valor anterior se reemplaza por el nuevo en la base de datos. Una alternativa es mantener las versiones de los elementos de datos a medida que se actualizan. Los algoritmos de este tipo se denominan control de concurrencia multiversión (MVCC). De esta forma, cada transacción "ve" el valor de un elemento de datos según su nivel de aislamiento. El control de concurrencia multiversión es otro intento de eliminar la sobrecarga de reinicio de las transacciones mediante el mantenimiento de múltiples versiones de elementos de datos y la programación de operaciones en la versión adecuada. La disponibilidad de múltiples versiones de la base de datos también permite consultas temporales que rastrean la evolución de los valores de los elementos de datos a lo largo del tiempo. Una preocupación del MVCC es la proliferación de múltiples versiones de elementos de datos actualizados. Para ahorrar espacio, las versiones de la base de datos pueden purgarse periódicamente. Esto debe hacerse cuando el SGBD distribuido esté seguro de que ya no recibirá una transacción que necesite acceder a las versiones purgadas.

Aunque la propuesta original data de 1978, ha ganado popularidad en los últimos años y ahora se implementa en diversos sistemas de gestión de bases de datos (SGBD) como IBM DB2, Oracle, SQL Server, SAP HANA, BerkeleyDB, PostgreSQL y sistemas como Spanner. Estos sistemas implementan el aislamiento de instantáneas, que se describe en la sección [5.3](#).

Las técnicas MVCC suelen utilizar marcas de tiempo para mantener el aislamiento de las transacciones, aunque existen propuestas que construyen el multiversionado sobre una capa de control de concurrencia basada en bloqueos. Aquí, nos centraremos en la implementación basada en marcas de tiempo que refuerza la serialización. En esta implementación, cada versión de un elemento de datos que se crea se etiqueta con la marca de tiempo de la transacción que lo creó. La idea es que cada operación de lectura acceda a la versión del elemento de datos correspondiente a su marca de tiempo, lo que reduce los abortos y reinicios de transacciones. Esto garantiza que

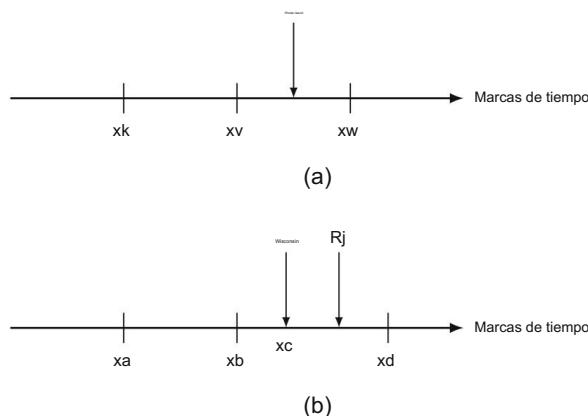


Fig. 5.8 Casos de TO multiversión

Cada transacción opera sobre un estado de la base de datos que habría visto si la transacción se hubiera ejecutado en serie en el orden de marca de tiempo.

La existencia de versiones es transparente para los usuarios que realizan transacciones simplemente haciendo referencia a elementos de datos, no a una versión específica. El gestor de transacciones asigna una marca de tiempo a cada transacción, que también se utiliza para registrar las marcas de tiempo de cada versión. Los historiales procesan las operaciones de la siguiente manera, garantizando un historial serializable:

1. Un $R_i(x)$ se traduce a una lectura en una versión de x . Esto se logra encontrando una versión de x (por ejemplo, x_v) tal que $ts(x_v)$ sea la marca de tiempo más grande menor que $ts(T_i)$. $R_i(x_v)$ se envía entonces al procesador de datos para leer x_v . Este caso se muestra en la Fig. 5.8a, que muestra que R_i puede leer la versión (x_v) que habría leído si hubiera llegado en el orden de la marca de tiempo.
2. Una $W_i(x)$ se traduce a $W_i(x_w)$ de modo que $ts(x_w) = ts(T_i)$ y se envía al procesador de datos solo si ninguna otra transacción con una marca de tiempo mayor que $ts(T_i)$ ha leído el valor de una versión de x (por ejemplo, x_r) tal que $ts(x_r) > ts(x_w)$. En otras palabras, si el planificador ya ha procesado una $R_j(x_r)$ tal que

$$ts(T_i) < ts(x_r) < ts(T_j)$$

Entonces, $W_i(x)$ se rechaza. Este caso se representa en la Fig. 5.8b, que muestra que si se acepta W_i , se crearía una versión (x_c) que R_j debería haber leído, pero no lo hizo, ya que no estaba disponible cuando se ejecutó R_j ; en su lugar, leyó la versión x_b , lo que genera un historial erróneo.

5.2.4 Algoritmos optimistas

Los algoritmos optimistas asumen que los conflictos de transacciones y la contención de datos No ser predominante y, por lo tanto, permitir que las transacciones se ejecuten sin sincronización hasta el final, cuando se valida su corrección. Optimista Los algoritmos de control de concurrencia pueden basarse en el bloqueo o en el sellado de tiempo, que fue la propuesta original. En esta sección, describimos un sistema distribuido basado en marcas de tiempo. algoritmo optimista.

Cada transacción sigue cinco fases: lectura (R), ejecución (E), escritura (W), validación (V) y confirmar (C); por supuesto, la fase de confirmación se convierte en aborto si la transacción es No validado. Este algoritmo asigna marcas de tiempo a las transacciones al inicio. de su paso de validación en lugar de al inicio de las transacciones como se hace con algoritmos TO (pesimistas). Además, no asocia lectura y escritura. marcas de tiempo con elementos de datos: solo funciona con marcas de tiempo de transacción durante la fase de validación.

Cada transacción T_i se subdivide (por el administrador de transacciones en el origen) sitio) en una serie de subtransacciones, cada una de las cuales puede ejecutarse en muchos sitios. Notacionalmente, denotemos por T una subtransacción de T_i que se ejecuta en el sitio s . En el Al comienzo de la fase de validación se asigna una marca de tiempo a la transacción, que También es la marca de tiempo de sus subtransacciones. Se realiza la validación local de T . de acuerdo con las siguientes reglas, que son mutuamente excluyentes.

Regla 1 En cada sitio s , si todas las transacciones T_k^s donde $ts(T_k^s) < ts(T_i^s)$ han completado su fase de escritura antes de T_i^s ha iniciado su fase de lectura (Fig. 5.9a), la validación tiene éxito, porque las ejecuciones de transacciones están en orden serial.

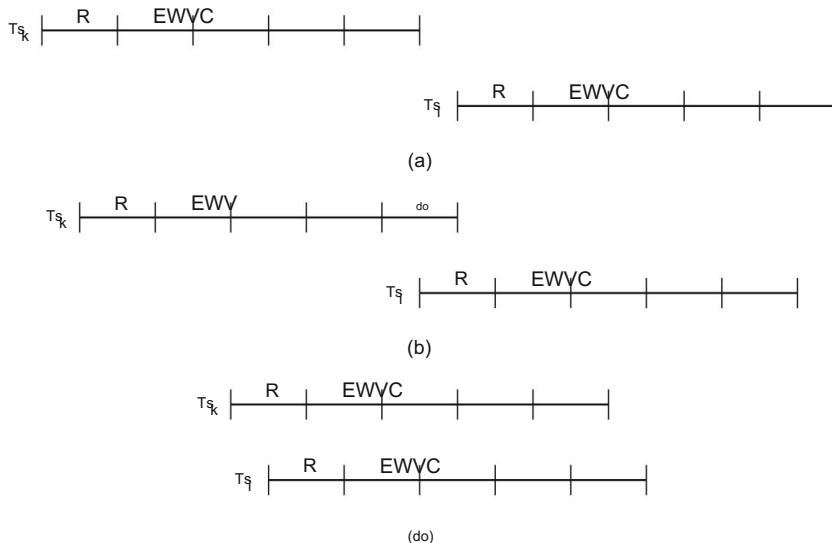


Fig. 5.9 Posibles escenarios de ejecución

Regla 2 En cada sitio s , si hay alguna transacción T y que $\frac{s}{k}$ tal que $ts(T) \frac{s}{k} < ts(T) \frac{s}{i}$, completa su fase de escritura mientras $T \frac{s}{i}$ está en su fase de lectura (Fig. 5.9b), el la validación tiene éxito si $WS(Tk) \cap RS(T) \frac{s}{i} = \cdot$.

Regla 3 En cada sitio s , si hay alguna transacción T que \xrightarrow{k}^s tal que $ts(T \xrightarrow{k}^s) < ts(T \xrightarrow{i}^s)$, y completa su fase de lectura antes de $T \xrightarrow{i}^s$ completa su fase de lectura (Fig. 5.9c), el la validación tiene éxito si $WS(T \xrightarrow{k}^s) \cap RS(T \xrightarrow{i}^s) = \emptyset$, y $WS(T \xrightarrow{k}^s) \cap WS(T \xrightarrow{i}^s) = \emptyset$.

La regla 1 es obvia: indica que las transacciones se ejecutan realmente en serie.

en su orden de marca de tiempo. La regla 2 garantiza que ninguno de los elementos de datos actualizados por $T_s^{\text{personalizado}}$ describirán actualizaciones en la base de datos (es decir,

comités) antes de T^s . Por lo tanto, las actualizaciones de T no se sobreescribirán

Por las actualizaciones de k^s , La regla 3 es similar a la regla 2, pero no requiere

Por las actualizaciones de k . La regla S es similar a la regla Z , pero no requiere que la escritura de T antes de que T comience a escribir. Simplemente requiere que las actualizaciones de T en la fase de lectura o la fase de escritura de T no afecten k .

Una vez que una transacción se valida localmente para garantizar que se mantenga la consistencia de la base de datos local, también debe validarse globalmente para garantizar que la coherencia mutua. Se cumple la regla de consistencia. Esto se logra asegurándose de que las reglas anteriores se cumplan en cada sitio participante.

Una ventaja de los algoritmos de control de concurrencia optimistas es el potencial de permitir un mayor nivel de concurrencia. Se ha demostrado que cuando la transacción

Los conflictos son muy raros, el mecanismo optimista funciona mejor que el bloqueo.

Una dificultad con los enfoques optimistas es el mantenimiento de la información.

Necesario para la validación. Para validar una subtransacción T, los conjuntos de

de transacciones terminadas que estaban en curso cuando T mantuvo^s. Llegué al sitio que necesitaba.

Otro problema es la inanición. Consideremos una situación en la que la fase de validación... de una transacción larga falla. En pruebas posteriores aún es posible que la validación fallará repetidamente. Por supuesto, es posible resolver este problema permitiendo la transacción acceso exclusivo a la base de datos después de un número específico de intentos. Sin embargo, esto reduce el nivel de concurrencia a una sola transacción. La exacta Una combinación de transacciones que causaría un nivel intolerable de reinicios es un problema que Queda por estudiar.

5.3 Control de concurrencia distribuida mediante instantáneas

Aislamiento

Hasta este punto, hemos analizado algoritmos que refuerzan la serialización. Aunque la serialización es el criterio de corrección más estudiado y discutido para la concurrencia en ejecución de transacciones, para algunas aplicaciones puede considerarse demasiado estricta en el sentido de que rechaza ciertas historias que podrían ser aceptables. En particular, la serialización crea un cuello de botella que impide que las bases de datos distribuidas escalen a grandes niveles. La razón principal es que restringe la concurrencia de transacciones.

muy intensamente, ya que las consultas de lectura grandes entran en conflicto con las actualizaciones. Esto ha llevado a... Definición de aislamiento de instantáneas (SI) como alternativa. El SI se ha adoptado ampliamente en sistemas comerciales y una serie de sistemas modernos, como Google Spanner y LeanXcale, que han logrado escalar a niveles muy grandes, dependen de él; lo discutimos Estos enfoques se describen en la sección 5.5. El aislamiento de instantáneas proporciona lecturas repetibles, pero no Aislamiento serializable. Cada transacción "ve" una instantánea consistente de la base de datos. cuando se inicia, y sus lecturas y escrituras se realizan en esta instantánea, por lo tanto, Las escrituras no son visibles para otras transacciones y no ve las escrituras de otras transacciones una vez que comienza a ejecutarse.

El aislamiento de instantáneas es un enfoque de múltiples versiones que permite que las transacciones lean la instantánea apropiada (es decir, la versión). Una ventaja importante del control de concurrencia basado en SI es que las transacciones de solo lectura pueden continuar sin un impacto significativo. Sobrecarga de sincronización. Para las transacciones de actualización, el algoritmo de control de concurrencia (en sistemas centralizados) es el siguiente:

- S1. Cuando se inicia una transacción T_i , obtiene una marca de tiempo de inicio $tsb(T_i)$.
- S2. Cuando T_i está listo para confirmar, obtiene una marca de tiempo de confirmación $tsc(T_i)$ que es mayor que cualquiera de los tsb o tsc existentes.
- S3. Ti confirma sus actualizaciones si no hay otro T_j tal que $tsc(T_j) < [tsb(T_i), tsc(T_i)]$ (es decir, no se ha confirmado ninguna otra transacción desde que comenzó T_i); De lo contrario, T_i se aborta. Esto se conoce como la regla del primer confirmador que gana, y evita la pérdida de actualizaciones.
- S4. Cuando se confirma T_i , sus cambios quedan disponibles para todas las transacciones T_k donde $tsb(T_k) > tsc(T_i)$.

Cuando se utiliza SI como criterio de corrección en el control de concurrencia distribuida, Un problema que debe abordarse es cómo calcular la instantánea consistente. (versión) en la que opera la transacción T_i . Si los conjuntos de lectura y escritura de la Si bien las transacciones se conocen de antemano, es posible calcular la instantánea de forma centralizada. (en el TM coordinador) mediante la recopilación de información de los sitios participantes. Esto, Por supuesto, no es realista. Lo que se necesita es una garantía global similar a la global. La garantía de serialización que comentamos antes. En otras palabras,

1. Cada historia local es SI, y
2. El historial global es SI, es decir, las órdenes de compromiso de las transacciones en cada sitio son lo mismo

Ahora identificamos las condiciones que deben cumplirse para la garantía anterior. para ser realizado. Comenzamos definiendo la relación de dependencia entre dos transacciones, lo cual es importante en este contexto ya que la instantánea que muestra una transacción Las lecturas de T_i deben incluir únicamente las actualizaciones de las transacciones de las que depende. transacción T_i en el sitio s (T_i^s) depende de T_j , denotado como dependiente ($T_i^s \rightarrow_j T_j^s$), si y sólo si $(RS(T_i^s) \cap WS(T_j^s)) = \emptyset$ ($WS(T_i^s) \cap RS(T_j^s) = \emptyset$) ($WS(T_i^s) \cap WS(T_j^s) = \emptyset$). Si hay algún sitio participante donde se cumple esta dependencia, entonces

Se cumple la condición dependiente (T_i, T_j).

Ahora estamos listos para especificar con mayor precisión las condiciones que deben cumplirse. Para garantizar la SI global, como se definió anteriormente. Las condiciones a continuación se dan para pares

transacciones, pero se cumplen transitivamente para un conjunto de transacciones. Para una transacción T_i para ver una instantánea globalmente consistente, se deben cumplir las siguientes condiciones para cada par de transacciones:

- C1. Si dependiente (T_i, T_j) $tsb(T_i) < tsc(T_j)$, entonces $tsb(T_i^s) < tsc(T_j^s)$ en cada sitio t donde T_i y T_j se ejecutan juntos.
- C2. Si depende (T_i, T_j) $tsc(T_i)$ sitio t donde $T_i^s < tsb(T_j)$, entonces $tsc(T_i^s) < tsb(T_j^s)$ en cada $y T_j$ se ejecutan juntos.
- C3. Si $tsc(T_i) < tsb(T_j)$, entonces $tsc(T_i^s) < tsb(T_j^s)$ en cada sitio t donde T_i y T_j ejecutar juntos.

Las dos primeras de estas condiciones garantizan que la función dependiente (T_i, T_j) sea verdadera en todo momento. sitios, es decir, T_i siempre ve correctamente esta relación entre sitios. La tercera condición garantiza que el orden de confirmación entre las transacciones sea el mismo en todos los sitios participantes, y evita que dos instantáneas incluyan confirmaciones parciales que son incompatibles con entre sí.

Antes de analizar el algoritmo de control de concurrencia de SI distribuido, identifiquemos La información que mantiene cada sitio:

- Para cualquier transacción activa T_i , el conjunto de transacciones activas y comprometidas en s se clasifican en dos grupos: aquellos que son concurrentes con T_i (es decir, cualquier T_j donde $tsb(T_i) < tsc(T_j)$) y aquellos que son seriales (es decir, cualquier T_j donde $tsc(T_i) < tsb(T_j)$)—tenga en cuenta que serial no es lo mismo que dependiente; la historia local indica ordenamiento en la historia local en s sin ninguna declaración sobre dependencia.
- Un reloj de eventos que aumenta monótonamente.

El algoritmo SI distribuido básico implementa básicamente el paso S3 del sistema centralizado. Algoritmo presentado anteriormente (aunque existen diferentes implementaciones), es decir, certifica si la transacción T_i puede confirmarse o debe cancelarse. El algoritmo

Procede de la siguiente manera:

- D1. El TM coordinador de T_i solicita a cada sitio participante que envíe su conjunto de transacciones concurrentes con T_i . A este mensaje se suma su propio evento reloj.
- D2. Cada sitio responde al TM coordinador con su conjunto local de transacciones concurrente con T_i .
- D3. La TM coordinadora fusiona todos los conjuntos de transacciones concurrentes locales en un conjunto de transacciones concurrentes globales para T_i .
- D4. La TM coordinadora envía esta lista global de transacciones concurrentes a todos los sitios participantes.
- D5. Cada sitio verifica si se cumplen las condiciones C1 y C2. Esto se hace mediante comprobar si hay una transacción T_j en la transacción concurrente global lista que está en la lista serial de la historia local (es decir, en la historia local de s , T_j tiene ejecutado antes de T_i), y del cual T_i depende (es decir, depende (T_i sostiene)). Si ese es el caso, T_i no ve una instantánea consistente en el sitio s , por lo que debe ser abortado. De lo contrario, T_i se valida en el sitio s .

D6. Cada sitio s envía su validación, positiva o negativa, a la TM coordinadora. Si se envía un mensaje de validación positiva, el sitio s actualiza su propio reloj de eventos al máximo de su propio reloj de eventos y del reloj de eventos de la TM coordinadora que recibió, y transfiere su nuevo valor de reloj al mensaje de respuesta.

D7. Si la TM coordinadora recibe un mensaje de validación negativo, Ti se cancela, ya que hay al menos un sitio donde no se visualiza una instantánea consistente. De lo contrario, la TM coordinadora certifica globalmente a Ti y le permite confirmar sus actualizaciones. Si se decide realizar una validación global, la TM coordinadora actualiza su propio reloj de eventos al máximo de los relojes de eventos que recibe de los sitios participantes y su propio reloj.

D8. La TM coordinadora informa a todos los sitios participantes que Ti está validado y puede confirmarse. También incorpora su nuevo valor de reloj de eventos, que se convierte en tsc(Ti).

D9. Al recibir este mensaje, cada sitio participante mantiene las actualizaciones de Ti y actualiza su propio reloj de eventos como antes.

En este algoritmo, la certificación de las condiciones C1 y C2 se realiza en el paso D5; los demás pasos sirven para recopilar la información necesaria y coordinar la verificación de la certificación. La sincronización del reloj de eventos entre los sitios permite aplicar la condición C3, garantizando que las órdenes de confirmación de las transacciones dependientes sean consistentes en todos los sitios, de modo que la instantánea global sea consistente.

El algoritmo que analizamos es un posible enfoque para implementar la seguridad de la información (SI) en un SGBD distribuido. Garantiza la SI global, pero requiere que la instantánea global se calcule por adelantado, lo que introduce varios cuellos de botella en la escalabilidad. Por ejemplo, enviar todas las transacciones concurrentes en el paso D2 obviamente no escala, ya que un sistema que ejecuta millones de transacciones concurrentes tendría que enviar millones de transacciones en cada verificación, lo que limitaría gravemente la escalabilidad. Además, requiere que todas las transacciones sigan el mismo proceso de certificación. Esto se puede optimizar separando las transacciones de un solo sitio que solo acceden a los datos en un sitio y, por lo tanto, no requieren la generación de una instantánea global de las transacciones globales que se ejecutan en varios sitios. Una forma de lograr esto es generar incrementalmente la instantánea que una transacción Ti lee a medida que se accede a los datos en diferentes sitios.

5.4 Confiabilidad del DBMS distribuido

En los SGBD centralizados, pueden ocurrir tres tipos de errores: fallos de transacción (p. ej., abortos de transacciones), fallos de sitio (que provocan la pérdida de datos en memoria, pero no en el almacenamiento persistente) y fallos de medios (que pueden causar la pérdida parcial o total de datos en el almacenamiento persistente). En un entorno distribuido, el sistema debe gestionar un cuarto tipo de fallo: los fallos de comunicación. Existen varios tipos de fallos de comunicación; los más comunes son los errores en los mensajes, mal ordenados.

Mensajes, mensajes perdidos (o no entregables) y fallos en la línea de comunicación. Los dos primeros errores son responsabilidad de la red informática y no los abordaremos más a fondo. Por lo tanto, en nuestros análisis de la fiabilidad de los SGBD distribuidos, esperamos que la red informática subyacente garantice que dos mensajes enviados desde un proceso en un sitio de origen a otro proceso en un sitio de destino se entreguen sin errores y en el orden en que se enviaron; es decir, consideramos cada enlace de comunicación como un canal FIFO fiable.

Los mensajes perdidos o no entregables suelen ser consecuencia de fallos en la línea de comunicación o en el sitio de destino. Si falla una línea de comunicación, además de perder los mensajes en tránsito, puede dividir la red en dos o más grupos separados. Esto se denomina partición de red. Si la red está particionada, los sitios de cada partición pueden seguir funcionando. En este caso, la ejecución de transacciones que acceden a datos almacenados en múltiples particiones se convierte en un problema importante. En un sistema distribuido, generalmente no es posible diferenciar los fallos del sitio de destino de los de las líneas de comunicación. En ambos casos, un sitio de origen envía un mensaje, pero no obtiene una respuesta dentro del tiempo previsto; esto se denomina tiempo de espera. En ese momento, los algoritmos de fiabilidad deben actuar.

Las fallas de comunicación señalan un aspecto único de las fallas en sistemas distribuidos. En sistemas centralizados, el estado del sistema puede caracterizarse como de todo o nada: el sistema está operativo o no. Por lo tanto, las fallas son completas: cuando una ocurre, todo el sistema deja de funcionar. Obviamente, esto no ocurre en sistemas distribuidos. Como ya hemos indicado, esta es su potencial punto fuerte. Sin embargo, también dificulta el diseño de algoritmos de gestión de transacciones, ya que, si un mensaje no se entrega, es difícil saber si el sitio receptor ha fallado o si se ha producido un fallo de red que impidió la entrega del mensaje.

Si los mensajes no se pueden entregar, asumimos que la red no hace nada al respecto. No los almacenará en búfer para su entrega al destino cuando se restablezca el servicio ni informará al proceso emisor de que el mensaje no se puede entregar. En resumen, el mensaje simplemente se perderá. Esta suposición se debe a que representa la menor expectativa de la red y asigna la responsabilidad de gestionar estos fallos al DBMS distribuido. En consecuencia, el DBMS distribuido es responsable de detectar la imposibilidad de entrega de un mensaje. El mecanismo de detección suele depender de las características del sistema de comunicación en el que se implementa el DBMS distribuido. Los detalles quedan fuera de nuestro alcance; en este análisis, asumiremos, como se mencionó anteriormente, que el emisor de un mensaje activará un temporizador y esperará hasta el final de un período de tiempo de espera para determinar que el mensaje no se ha entregado.

Los protocolos de confiabilidad distribuida buscan mantener la atomicidad y durabilidad de las transacciones distribuidas que se ejecutan en varias bases de datos. Estos protocolos abordan la ejecución distribuida de los comandos Begin_transaction, Read, Write, Abort, Commit y recovery . Los comandos Begin_transaction, Read y Write son ejecutados por los gestores de recuperación local (LRM) de la misma manera que en los sistemas de gestión de bases de datos (SGBD) centralizados. Los comandos que requieren especial cuidado en los SGBD distribuidos son los comandos Commit, Abort y Read . La dificultad fundamental radica en garantizar que todos..

de los sitios que participan en la ejecución de una transacción llegan a la misma decisión (abortar o confirmar) respecto al destino de la transacción.

La implementación de protocolos de confiabilidad distribuida dentro del modelo arquitectónico adoptado en este libro plantea diversas cuestiones interesantes y complejas. Las abordamos en la sección 5.4.6 , tras la introducción de los protocolos. Por el momento, adoptamos una abstracción común: asumimos que en el punto de origen de una transacción existe un proceso coordinador y que en cada punto donde se ejecuta la transacción existen procesos participantes . Por lo tanto, los protocolos de confiabilidad distribuida se implementan entre el coordinador y los participantes.

Las técnicas de confiabilidad en sistemas de bases de datos distribuidas consisten en protocolos de confirmación, terminación y recuperación. Los protocolos de confirmación y recuperación especifican cómo se ejecutan los comandos Commit y recovery , mientras que los protocolos de terminación especifican cómo los sitios activos pueden terminar una transacción cuando detectan un fallo. Los protocolos de terminación y recuperación son dos caras opuestas del problema de la recuperación: ante un fallo en un sitio, los protocolos de terminación abordan cómo los sitios operativos gestionan el fallo, mientras que los protocolos de recuperación abordan el procedimiento que debe seguir el proceso (coordinador o participante) en un sitio fallido para recuperar su estado una vez que el sitio se reinicia. En el caso de la partición de la red, los protocolos de terminación toman las medidas necesarias para terminar las transacciones activas que se ejecutan en diferentes particiones, mientras que los protocolos de recuperación abordan cómo se restablece la consistencia global de la base de datos tras la reconexión de las particiones de la red.

El requisito principal de los protocolos de confirmación es que mantengan la atomicidad de las transacciones distribuidas. Esto significa que, aunque la ejecución de la transacción distribuida involucra múltiples sitios, algunos de los cuales podrían fallar durante la ejecución, el impacto de la transacción en la base de datos distribuida es de todo o nada.

Esto se denomina compromiso atómico. Preferimos que los protocolos de terminación sean no bloqueantes. Un protocolo es no bloqueante si permite que una transacción termine en los sitios operativos sin esperar la recuperación del sitio fallido. Esto mejoraría significativamente el tiempo de respuesta de las transacciones. También nos gustaría que los protocolos de recuperación distribuida fueran independientes. Los protocolos de recuperación independientes determinan cómo terminar una transacción que se estaba ejecutando en el momento del fallo sin tener que consultar ningún otro sitio. La existencia de estos protocolos reduciría la cantidad de mensajes que deben intercambiarse durante la recuperación. Cabe destacar que la existencia de protocolos de recuperación independientes implicaría la existencia de protocolos de terminación no bloqueantes, pero no a la inversa.

5.4.1 Protocolo de confirmación de dos fases

La confirmación en dos fases (2PC) es un protocolo muy simple y elegante que garantiza la confirmación atómica de transacciones distribuidas. Extiende los efectos de las acciones de confirmación atómica local a las transacciones distribuidas al exigir que todos los sitios involucrados en la ejecución de una transacción distribuida acepten confirmar la transacción antes de su...

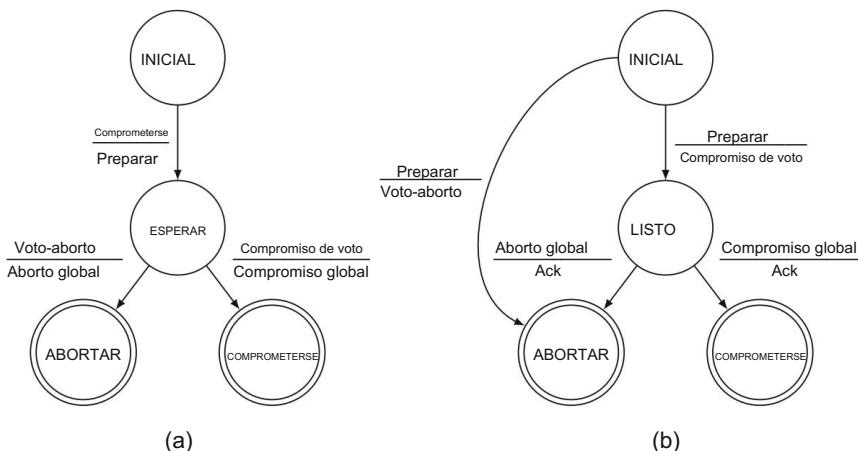


Fig. 5.10 Transiciones de estado en el protocolo 2PC. (a) Estados coordinadores. (b) Estados participantes.

Los efectos se hacen permanentes. Existen varias razones por las que dicha sincronización entre sitios es necesaria. En primer lugar, dependiendo del tipo de algoritmo de control de concurrencia utilizado, algunos programadores podrían no estar listos para finalizar una transacción.

Por ejemplo, si una transacción ha leído el valor de un dato actualizado por otra transacción aún no confirmada, es posible que el programador asociado no desee confirmar el primero. Por supuesto, los algoritmos de control de concurrencia estrictos que evitan abortos en cascada no permitirían que otra transacción lea el valor actualizado de un dato hasta que finalice la transacción de actualización. Esto se denomina a veces condición de recuperabilidad.

Otra posible razón por la que un participante podría no aceptar comprometerse se debe a interbloqueos que requieren que el participante cancele la transacción. Tenga en cuenta que, en este caso, se debe permitir al participante cancelar la transacción sin que se le indique. Esta capacidad es muy importante y se denomina cancelación unilateral. Otra razón para la cancelación unilateral puede ser el tiempo de espera, como se mencionó anteriormente.

A continuación se presenta una breve descripción del protocolo 2PC que no considera fallos. Este análisis se facilita mediante el diagrama de transición de estados del protocolo 2PC (Fig. 5.10). Los estados se representan mediante círculos y los bordes representan las transiciones de estado. Los estados terminales se representan mediante círculos concéntricos. La interpretación de las etiquetas en los bordes es la siguiente: el motivo de la transición de estado, que es un mensaje recibido, se indica en la parte superior, y el mensaje enviado como resultado de la transición de estado, en la parte inferior.

1. Inicialmente, el coordinador escribe un registro begin_commit en su registro, envía un mensaje de "preparación" a todos los sitios participantes y entra en el estado de ESPERA.
2. Cuando un participante recibe un mensaje de "preparación", verifica si puede confirmar la transacción. De ser así, el participante escribe un registro de "listo" en el registro, envía un mensaje de "voto de confirmación" al coordinador y entra en estado LISTO; de lo contrario,

El participante escribe un registro de aborto y envía un mensaje de “voto-aborto” al coordinador.

3. Si la decisión del sitio es abortar, puede olvidarse de esa transacción, ya que una decisión de abortar sirve como voto (es decir, aborto unilateral).
4. Tras recibir la respuesta de todos los participantes, el coordinador decide si confirma o cancela la transacción. Si un solo participante ha registrado un voto negativo, el coordinador debe cancelar la transacción globalmente.
Por lo tanto, escribe un registro de aborto , envía un mensaje de “aborto global” a todos los sitios participantes y entra en el estado ABORT; de lo contrario, escribe un registro de confirmación , envía un mensaje de “commit global” a todos los participantes y entra en el estado COMMIT.
5. Los participantes confirman o abortan la transacción según las instrucciones del coordinador y envían un acuse de recibo, momento en el cual el coordinador finaliza la transacción escribiendo un registro de fin de transacción en el registro.

Observe cómo el coordinador toma una decisión de terminación global respecto a una transacción. Dos reglas rigen esta decisión, que, en conjunto, se denominan regla de confirmación global:

1. Si incluso un participante vota para abortar la transacción, el coordinador debe llegar a una decisión global de abortar.
2. Si todos los participantes votan para confirmar la transacción, el coordinador debe llegar a una decisión de compromiso global.

El funcionamiento del protocolo 2PC entre un coordinador y un participante en ausencia de fallos se representa en la figura 5.11, donde los círculos indican los estados y las líneas discontinuas indican los mensajes entre el coordinador y los participantes.

Las etiquetas en las líneas discontinuas especifican la naturaleza del mensaje.

Algunos puntos importantes sobre el protocolo 2PC que se pueden observar en la Fig. 5.11 son los siguientes. Primero, 2PC permite a un participante abortar unilateralmente una transacción hasta que haya registrado un voto afirmativo. Segundo, una vez que un participante vota para confirmar o abortar una transacción, no puede cambiar su voto. Tercero, mientras un participante está en el estado LISTO, puede abortar o confirmar la transacción, dependiendo de la naturaleza del mensaje del coordinador. Cuarto, la decisión de terminación global la toma el coordinador según la regla de confirmación global. Finalmente, cabe destacar que los procesos coordinador y participante entran en ciertos estados donde deben esperar mensajes mutuos. Para garantizar que puedan salir de estos estados y terminar, se utilizan temporizadores. Cada proceso activa su temporizador al entrar en un estado, y si no se recibe el mensaje esperado antes de que se agote el temporizador, el proceso expira e invoca su protocolo de tiempo de espera (que se explicará más adelante).

Existen diversas maneras de implementar un protocolo 2PC. El protocolo descrito anteriormente, que se muestra en la Fig. 5.11 , se denomina 2PC centralizado , ya que la comunicación se establece únicamente entre el coordinador y los participantes; estos no se comunican entre sí. Esta estructura de comunicación, que constituye la base de nuestros análisis posteriores, se ilustra con mayor claridad en la Fig. 5.12.

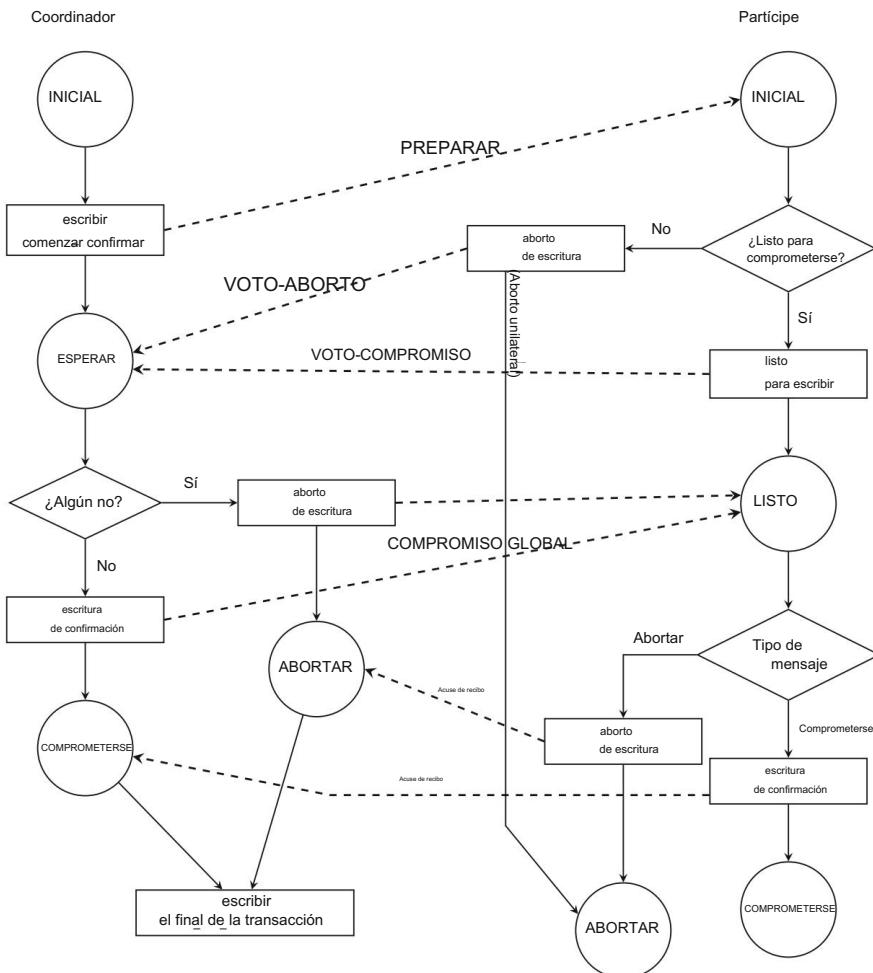


Fig. 5.11 Acciones del protocolo 2PC

Otra alternativa es el protocolo 2PC lineal (también llamado 2PC anidado), donde los participantes pueden comunicarse entre sí. Existe un posible orden lógico entre los sitios del sistema para fines de comunicación. Supongamos que el orden entre los sitios que participan en la ejecución de una transacción es 1, ..., N, donde el coordinador es el primero en el orden. El protocolo 2PC se implementa mediante una comunicación directa del coordinador (número 1) a N, durante la cual se completa la primera fase, y mediante una comunicación inversa de N al coordinador, durante la cual se completa la segunda fase. Por lo tanto, el protocolo 2PC lineal funciona de la siguiente manera.

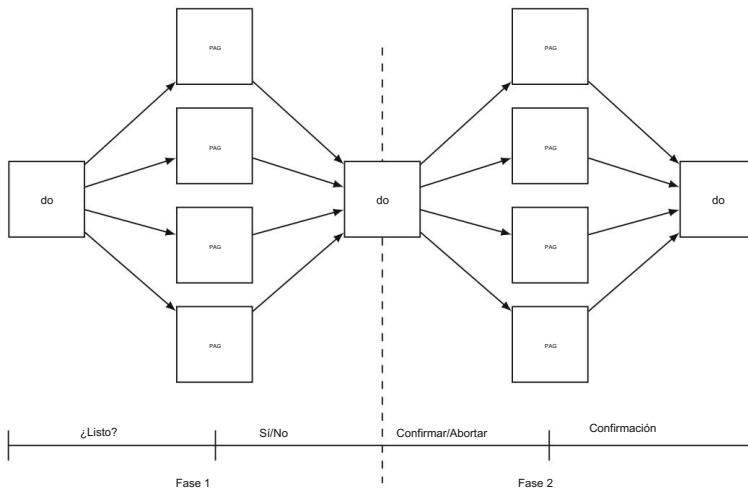


Fig. 5.12 Estructura de comunicación centralizada 2PC. C: Coordinador, P: Participante

El coordinador envía el mensaje “preparar” al participante 2. Si el participante 2 no está listo para confirmar la transacción, envía un mensaje “votar-abortar” (VA) al participante 3 y la transacción se aborta en este punto (aborts unilateral por parte de 2). Si, por otro lado, el participante 2 acepta confirmar la transacción, envía un mensaje de confirmación (VC) al participante 3 y entra en el estado LISTO. Al igual que en la implementación centralizada de 2PC, cada sitio registra su decisión antes de enviar el mensaje al siguiente. Este proceso continúa hasta que el participante N recibe una confirmación. Este es el final de la primera fase. Si el sitio N decide confirmar, envía de vuelta al sitio (N – 1) una confirmación global (GC); de lo contrario, envía un mensaje de cancelación global (GA). En consecuencia, los participantes entran en el estado correspondiente (COMMIT o ABORT) y propagan el mensaje de vuelta al coordinador.

El 2PC lineal, cuya estructura de comunicación se muestra en la Fig. 5.13, genera menos mensajes, pero no proporciona paralelismo. Por lo tanto, presenta un bajo tiempo de respuesta.

Otra estructura de comunicación popular para la implementación del protocolo 2PC implica la comunicación entre todos los participantes durante la primera fase del protocolo, de modo que cada uno tome decisiones de terminación de forma independiente respecto a la transacción específica. Esta versión, denominada 2PC distribuido, elimina la necesidad de la segunda fase del protocolo, ya que los participantes pueden tomar una decisión por sí mismos. Funciona de la siguiente manera: el coordinador envía el mensaje de preparación a todos los participantes. Cada participante envía su decisión a los demás participantes (y al coordinador) mediante un mensaje de “voto confirmado” o de “voto abortado”. Cada participante espera los mensajes de todos los demás participantes y toma su decisión de terminación según la regla de compromiso global. Obviamente, no es necesaria la segunda fase del protocolo (que alguien envíe la decisión de aborto global o compromiso global a los demás), ya que cada participante tiene...

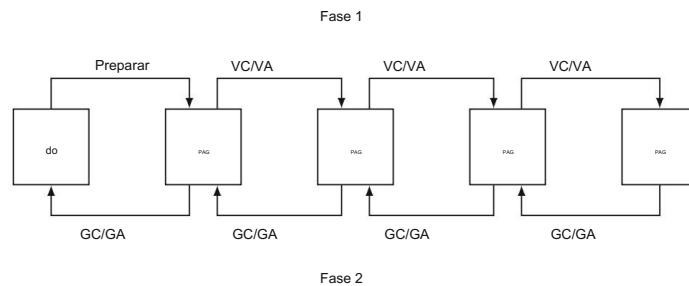


Fig. 5.13 Estructura de comunicación lineal 2PC. VC: vote.commit; VA: vote.abort; GC: global.commit; GA: global.abort

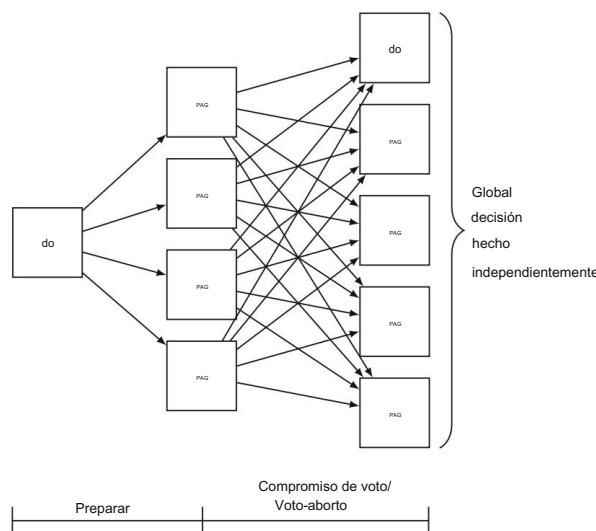


Fig. 5.14 Estructura de comunicación distribuida 2PC

Se llegó a esa decisión al final de la primera fase. La estructura de comunicación

El proceso de confirmación distribuida se muestra en la figura 5.14.

En la implementación 2PC lineal y distribuida, un participante debe conocer la identidad del siguiente participante en el ordenamiento lineal (en caso de 2PC lineal) o de todos los participantes (en caso de 2PC distribuido). Este problema se puede solucionar adjuntando la lista de participantes al mensaje de preparación que envía el coordinador. Este problema no se plantea en el caso de 2PC centralizado, ya que el coordinador sabe claramente quiénes son los participantes.

El algoritmo para la ejecución centralizada del protocolo 2PC por parte de la coordinación Los resultados de los experimentos y de los participantes se dan en los algoritmos 5.6 y 5.7, respectivamente.

Algoritmo 5.6: Coordinador 2PC (2PC-C)

```

comenzar
    repetir esperar un evento
        cambiar evento hacer
            caso Msg Llegada hacer
                Deje que el mensaje que llegó sea msg
                switch msg do case
                    Commit do           {comando de confirmación del programador}
                        escribir registro begin_commit en el registro enviar
                        mensaje "Preparado" a todos los participantes involucrados establecer
                        temporizador
                    finalizar
                    caso caso Votación-aborto hacer {un participante ha votado para abortar; unilateral
                        abortar}
                        escribir registro de aborto en el
                        registro enviar mensaje "Global-abort" a los otros participantes involucrados
                        establecer
                        temporizador
                    fin caso caso Votar-confirmar
                        actualizar la lista de participantes que han respondido si todos los
                        participantes han respondido entonces {todos deben haber votado para
                        confirmar} escribir
                            registro de confirmación en el registro
                            enviar mensaje "Global-commit" a todos los participantes involucrados
                            establecer
                            temporizador fin si
                    Fin del caso
                    caso Ack hacer
                        actualizar la lista de participantes que han reconocido si todos los
                        participantes han reconocido entonces escribir el registro
                            |   end_of_transaction en el registro de lo contrario enviar
                            |
                            decisión global a los participantes que no respondieron fin si
                    fin de caso
                    caso Tiempo de
                        |   espera para ejecutar el protocolo de
                        terminación fin de caso
                    interruptor final
                fin de caso
            interruptor final
        hasta el fin de los
        tiempos
    
```

5.4.2 Variaciones de 2PC

Se han propuesto dos variantes de 2PC para mejorar su rendimiento. Esto se logra reduciendo (1) el número de mensajes transmitidos entre el coordinador y los participantes, y (2) la cantidad de registros que se escriben.

Algoritmo 5.7: Participante 2PC (2PC-P)

```

comenzar
    repetir esperar un
        evento cambiar
            ev hacer caso Msg Llegada hacer
                Deje que el mensaje que llegó sea msg
                switch msg do case
                    Prepare do          {Preparar comando del coordinador} Si está listo para
                        confirmar , escriba el registro
                        de listo en el registro. Envíe el
                        mensaje "Votar-confirmar" al coordinador. Establezca el fin del
                        temporizador
                        si no.
                            {abortedo unilateral}
                            escribir registro de aborto en el
                            registro enviar mensaje "Voto-abort" al coordinador abortar
                            la transacción finalizar si
                    fin de caso
                    caso Global-abort escribir
                        registro de aborto en el registro
                        abortar la transacción fin
                    de caso
                    caso Global-commit escribir
                        registro de confirmación en el registro
                        confirmar la transacción fin de
                    caso
                    interruptor final
                fin de caso
                caso Tiempo de
                    |   espera para ejecutar el protocolo de
                    |   terminación fin de caso
                interruptor final
            hasta el fin de los
            tiempos
        
```

Estos protocolos se denominan aborto presunto y confirmación presuntuosa. El aborto presuntuoso está optimizado para gestionar transacciones de solo lectura, así como transacciones de actualización, algunos de cuyos procesos no realizan actualizaciones en la base de datos (denominadas parcialmente de solo lectura). El protocolo de confirmación presuntuosa está optimizado para gestionar las transacciones de actualización generales. Analizaremos brevemente ambas variantes

5.4.2.1 Protocolo de aborto presunto 2PC

En el protocolo de aborto presunto 2PC, cuando un participante preparado consulta al coordinador sobre el resultado de una transacción y no hay información al respecto, la respuesta a la consulta es abortar la transacción. Esto funciona porque, en el caso de

una confirmación, el coordinador no se olvida de una transacción hasta que todos los participantes la reconocen, garantizando que ya no volverán a preguntar sobre esta transacción.

Al usar esta convención, el coordinador puede olvidarse de una transacción inmediatamente después de decidir abortarla. Puede escribir un registro de abortar sin esperar que los participantes confirmen la orden. El coordinador no necesita escribir un registro de fin de transacción después de un registro de abortar .

No es necesario forzar el registro de cancelación , ya que si un sitio falla antes de recibir la decisión y luego se recupera, la rutina de recuperación revisará el registro para determinar el destino de la transacción. Dado que el registro de cancelación no se fuerza, la rutina de recuperación podría no encontrar información sobre la transacción; en ese caso, consultará al coordinador y se le indicará que la cancele. Por la misma razón, los participantes tampoco necesitan forzar los registros de cancelación .

Dado que ahorra cierta transmisión de mensajes entre el coordinador y los participantes en caso de transacciones abortadas, se espera que la supuesta interrupción 2PC sea más eficiente.

5.4.2.2 Protocolo de compromiso presunto 2PC

La confirmación presunta 2PC se basa en la premisa de que si no existe información sobre la transacción, esta debe considerarse confirmada. Sin embargo, no es un dual exacto de la cancelación presunta 2PC, ya que un dual exacto requeriría que el coordinador olvide una transacción inmediatamente después de decidir confirmarla, que no se fuercen los registros de confirmación (incluidos los registros listos de los participantes) y que no sea necesario confirmar los comandos de confirmación. Considere, sin embargo, el siguiente escenario.

El coordinador envía mensajes preparados y comienza a recopilar información, pero falla antes de poder recopilarla toda y tomar una decisión. En este caso, los participantes esperarán hasta que se agote el tiempo de espera y luego transferirán la transacción a sus rutinas de recuperación. Al no haber información sobre la transacción, las rutinas de recuperación de cada participante la confirmarán. El coordinador, por otro lado, abortará la transacción al recuperarse, lo que provocará una inconsistencia.

La confirmación presunta 2PC resuelve el problema de la siguiente manera. El coordinador, antes de enviar el mensaje de preparación, fuerza la escritura de un registro de recopilación que contiene los nombres de todos los participantes involucrados en la ejecución de la transacción. El participante entra entonces en el estado de RECOLECCIÓN, tras lo cual envía el mensaje de preparación y entra en el estado de ESPERA. Los participantes, al recibir el mensaje de preparación, deciden qué hacer con la transacción, escriben un registro de cancelación/listo según corresponda y responden con un mensaje de "voto de cancelación" o de "voto de confirmación". Cuando el coordinador recibe las decisiones de todos los participantes, decide cancelar o confirmar la transacción. Si decide cancelar, el coordinador escribe un registro de cancelación , entra en el estado de ABORTO y envía un mensaje de "aborted global". Si decide confirmar la transacción, escribe un registro de confirmación , envía un comando de "confirmación global" y olvida la transacción. Cuando los participantes reciben un mensaje de confirmación global, escriben un registro de confirmación y actualizan la base de datos. Si reciben un mensaje de cancelación global, escriben un registro de cancelación y...

Reconocimiento. El participante, al recibir el acuse de recibo de la interrupción, escribe un registro de fin de transacción y olvida la transacción.

5.4.3 Manejo de fallas del sitio

En esta sección, analizamos el fallo de los sitios en la red. Nuestro objetivo es desarrollar protocolos de terminación sin bloqueo y de recuperación independiente. Como se indicó anteriormente, la existencia de protocolos de recuperación independientes implicaría la existencia de protocolos de recuperación sin bloqueo. Sin embargo, nuestro análisis aborda ambos aspectos por separado. Cabe destacar que, en el siguiente análisis, solo consideramos el protocolo 2PC estándar, no sus dos variantes presentadas anteriormente.

Primero, definamos los límites para la existencia de protocolos de terminación no bloqueantes y de recuperación independientes ante fallos de sitio. Se ha demostrado que dichos protocolos existen cuando falla un solo sitio. Sin embargo, en caso de fallos de varios sitios, las perspectivas son menos prometedoras. Un resultado negativo indica que no es posible diseñar protocolos de recuperación independientes (y, por lo tanto, protocolos de terminación no bloqueantes) cuando fallan varios sitios. Primero, desarrollamos protocolos de terminación y recuperación para el algoritmo 2PC y demostramos que este algoritmo es inherentemente bloqueante.

Luego procedemos al desarrollo de protocolos de confirmación atómica que no son bloqueantes en el caso de fallas de un solo sitio.

5.4.3.1 Protocolos de terminación y recuperación para 2PC

Protocolos de terminación

Los protocolos de terminación gestionan los tiempos de espera tanto para los procesos coordinadores como para los participantes. Un tiempo de espera se produce en un sitio de destino cuando no puede recibir un mensaje esperado de un sitio de origen dentro del plazo previsto. En esta sección, consideraremos que esto se debe a un fallo del sitio de origen.

El método para gestionar los tiempos de espera depende del momento en que se producen los fallos, así como de su tipo. Por lo tanto, es necesario considerar los fallos en varios puntos de la ejecución de 2PC. A continuación, se hace referencia al diagrama de transición de estados de 2PC (Fig. 5.10).

Tiempos de espera del coordinador

Hay tres estados en los que el coordinador puede agotar el tiempo de espera: WAIT, COMMIT y ABORT. Los tiempos de espera en los dos últimos estados se gestionan de la misma manera. Por lo tanto, solo debemos considerar dos casos:

1. Tiempo de espera agotado en el estado de espera. En este estado, el coordinador espera las decisiones locales de los participantes. El coordinador no puede confirmar la transacción unilateralmente, ya que no se ha cumplido la regla de confirmación global. Sin embargo, puede decidir cancelar la transacción globalmente, en cuyo caso escribe un registro de cancelación en el registro y envía un mensaje de "cancelación global" a todos los participantes.
2. Tiempo de espera en los estados COMMIT o ABORT. En este caso, el coordinador no está seguro de que los administradores de recuperación locales hayan completado los procedimientos de confirmación o cancelación en todos los sitios participantes. Por lo tanto, el coordinador envía repetidamente los comandos "global-commit" o "global-abort" a los sitios que aún no han respondido y espera su confirmación.

Tiempos de espera de los participantes

Un participante puede agotar el tiempo de espera en dos estados: INICIAL y LISTO. Examinemos Ambos casos.

1. Tiempo de espera en el estado INICIAL. En este estado, el participante espera un mensaje de "preparación". El coordinador debe haber fallado en el estado INICIAL. El participante puede cancelar la transacción unilateralmente tras un tiempo de espera. Si el mensaje de "preparación" llega posteriormente, puede gestionarse de dos maneras: el participante consultaría su registro, encontraría el registro de cancelación y respondería con un "voto de cancelación", o simplemente ignoraría el mensaje de "preparación". En este último caso, el coordinador se quedaría en estado de espera y seguiría el procedimiento descrito anteriormente.
2. Tiempo de espera en el estado LISTO. En este estado, el participante votó para confirmar la transacción, pero desconoce la decisión global del coordinador. El participante no puede tomar una decisión unilateralmente. Al estar en el estado LISTO, debe haber votado para confirmar la transacción. Por lo tanto, no puede cambiar su voto y abortarla unilateralmente. Por otro lado, no puede decidir confirmarla unilateralmente, ya que es posible que otro participante haya votado para abortarla. En este caso, el participante permanecerá bloqueado hasta que alguien (ya sea el coordinador u otro participante) le informe sobre el destino final de la transacción.

Consideraremos una estructura de comunicación centralizada donde los participantes no pueden comunicarse entre sí. En este caso, el participante que intenta finalizar una transacción debe solicitar la decisión del coordinador y esperar una respuesta. Si el coordinador falla, el participante permanecerá bloqueado. Esto no es deseable.

En algunas discusiones sobre el protocolo 2PC, se asume que los participantes no usan temporizadores ni agotan el tiempo de espera. Sin embargo, implementar protocolos de tiempo de espera para los participantes resuelve algunos problemas problemáticos y puede acelerar el proceso de confirmación. Por lo tanto, consideraremos este caso más general.

Si los participantes pueden comunicarse entre sí, se podría desarrollar un protocolo de terminación más distribuido. El participante que agota el tiempo de espera puede simplemente pedir a los demás participantes que le ayuden a tomar una decisión. Suponiendo que el participante P_i es el que agota el tiempo de espera, cada uno de los demás participantes (P_j) responde de la siguiente manera:

1. P_j se encuentra en estado INICIAL. Esto significa que P_j aún no ha votado y es posible que ni siquiera haya recibido el mensaje de "preparación". Por lo tanto, puede cancelar la transacción unilateralmente y responder a P_i con un mensaje de "votación-cancelación".
2. P_j está en estado LISTO. En este estado, P_j ha votado para confirmar la transacción, pero no ha recibido ninguna notificación sobre la decisión global. Por lo tanto, no puede ayudar a P_i a finalizar la transacción.
3. P_j se encuentra en los estados ABORT o COMMIT. En estos estados, P_j ha decidido unilateralmente abortar la transacción o ha recibido la decisión del coordinador sobre la terminación global. Por lo tanto, puede enviar a P_i un mensaje de "vote-commit" o de "vote-abort".

Piense en cómo el participante que pierde el tiempo (P_i) puede interpretar estas respuestas.

Son posibles los siguientes casos:

1. P_i recibe mensajes de "votación-aborto" de todos los P_j . Esto significa que ninguno de los demás participantes había votado aún, pero decidieron cancelar la transacción unilateralmente. En estas circunstancias, P_i puede proceder a cancelar la transacción.
2. P_i recibe mensajes de "voto-aborto" de algunos P_j , pero otros participantes indican que están en estado LISTO. En este caso, P_i aún puede proceder y abortar la transacción, ya que, según la regla de confirmación global, la transacción no puede confirmarse y eventualmente será abortada.
3. P_i recibe una notificación de todos los P_j indicando que están en estado LISTO. En este caso, ninguno de los participantes tiene suficiente información sobre el destino de la transacción como para finalizarla correctamente.
4. P_i recibe mensajes de "aborted global" o "confirmation global" de todos los P_j . En este caso, todos los demás participantes han recibido la decisión del coordinador. Por lo tanto, P_i puede proceder y finalizar la transacción según los mensajes que reciba de los demás participantes. Cabe destacar que no es posible que algunos participantes respondan con un "aborted global" mientras que otros respondan con un "compromiso global", ya que esto no puede ser el resultado de una ejecución legítima del protocolo 2PC.
5. P_i recibe un mensaje de "aborted global" o "confirmation global" de algunos P_j , mientras que otros indican que están en estado LISTO. Esto indica que algunos sitios han recibido la decisión del coordinador, mientras que otros aún la esperan. En este caso, P_i puede proceder como en el caso 4 anterior.

Estos cinco casos abarcan todas las alternativas que un protocolo de terminación debe gestionar. No es necesario considerar casos en los que, por ejemplo, un participante envía un mensaje de "votación-aborto" mientras otro envía un mensaje de "confirmación global". Esto no puede ocurrir en 2PC. Durante la ejecución del protocolo 2PC, ningún proceso (participante o coordinador) se encuentra a más de una transición de estado de cualquier otro proceso.

Por ejemplo, si un participante está en el estado INICIAL, todos los demás participantes están en

El estado inicial o listo. De igual forma, el coordinador se encuentra en el estado inicial o en espera. Por lo tanto, se dice que todos los procesos en un protocolo 2PC son síncronos dentro de una transición de estado.

Tenga en cuenta que, en el caso 3, los procesos participantes permanecen bloqueados, ya que no pueden finalizar una transacción. En ciertas circunstancias, puede haber una manera de superar este bloqueo. Si durante la finalización todos los participantes se dan cuenta de que solo el sitio coordinador ha fallado, pueden elegir un nuevo coordinador, lo que puede reiniciar el proceso de confirmación. Existen diferentes maneras de elegir al coordinador. Es posible definir un orden total entre todos los sitios y elegir al siguiente en orden, o establecer un procedimiento de votación entre los participantes.

Sin embargo, esto no funcionará si

fallan tanto el sitio participante como el sitio coordinador. En este caso, es posible que el participante en el sitio fallido haya recibido la decisión del coordinador y haya terminado la transacción en consecuencia. Esta decisión es desconocida para los demás participantes; por lo tanto, si eligen un nuevo coordinador y proceden, existe el riesgo de que decidan terminar la transacción de forma diferente a la del participante en el sitio fallido. Es evidente que no es posible diseñar protocolos de terminación para 2PC que garanticen una terminación sin bloqueo. Por lo tanto, el protocolo 2PC es un protocolo de bloqueo. Formalmente, el protocolo es de bloqueo porque hay un estado en la Fig. 5.10 adyacente tanto al estado de confirmación como al de cancelación, y cuando hay una falla del coordinador, los participantes están en estado listo. Por lo tanto, es imposible determinar si el coordinador pasó al estado de cancelación o confirmación hasta que se recupere.

El protocolo 3PC (confirmación de tres fases) resuelve esta situación de bloqueo agregando un nuevo estado, PRECOMMIT, entre los estados de espera y confirmación para evitar la situación y prevenir la situación de bloqueo en caso de una falla del coordinador.

Dado que asumimos una estructura de comunicación centralizada al desarrollar los algoritmos 2PC en los algoritmos 5.6 y 5.7, continuaremos con la misma suposición al desarrollar los protocolos de terminación. La parte del código que debe incluirse en la sección de tiempo de espera de los algoritmos 2PC del coordinador y del participante se proporciona en los algoritmos 5.8 y 5.9, respectivamente.

Algoritmo 5.8: Terminación del coordinador 2PC

```

comenzar
    si está en estado de ESPERA entonces
        | escribir el registro de aborto en el
        | registro envía el mensaje "Aborto global" a todos los participantes
        | {El coordinador está en estado ABORT}
    de
        | lo contrario verifica el último registro
        | si el último registro = aborto entonces
            |   | enviar "Global-abort" a todos los participantes que no han respondido
            |   demás
            |   | enviar "Global-commit" a todos los participantes que no han respondido si
        | finalizar si se
        | establece el temporizador
fin

```

Algoritmo 5.9: 2PC-Participante Terminado

```

comenzar
    si está en estado INICIAL entonces
        |   escribir registro de aborto en el registro
    de lo
        |   contrario enviar mensaje de "Votación confirmada" al coordinador
        |   reiniciar
    temporizador fin si
fin

```

Protocolos de recuperación

En la sección anterior, analizamos cómo el protocolo 2PC gestiona las fallas desde la perspectiva de los sitios operativos. En esta sección, adoptamos la perspectiva opuesta: nos interesa investigar los protocolos que un coordinador o participante puede usar para recuperar sus estados cuando sus sitios fallan y luego se reinician. Cabe recordar que deseamos que estos protocolos sean independientes. Sin embargo, en general, no es posible diseñar protocolos que garanticen una recuperación independiente manteniendo la atomicidad de las transacciones distribuidas. Esto no es sorprendente, dado que los protocolos de terminación para 2PC son inherentemente bloqueantes.

En la siguiente discusión, utilizamos nuevamente el diagrama de transición de estados de la Figura 5.10. Además, hacemos dos suposiciones interpretativas: (1) la acción combinada de escribir un registro en el registro y enviar un mensaje se asume como átomica, y (2) la transición de estado ocurre después de la transmisión del mensaje de respuesta. Por ejemplo, si el coordinador está en estado de espera, esto significa que ha escrito correctamente el registro begin_commit en su registro y ha transmitido correctamente el comando "pre-prepare". Sin embargo, esto no indica nada sobre la finalización exitosa de la transmisión del mensaje. Por lo tanto, el mensaje "pre-prepare" podría no llegar nunca a los participantes debido a fallas de comunicación, que analizamos por separado. La primera suposición relacionada con la atomicidad es, por supuesto, poco realista. Sin embargo, simplifica nuestro análisis de los casos de falla fundamental. Al final de esta sección, mostramos que

Los demás casos que surgen de la relajación de este supuesto pueden manejarse mediante una combinación de los casos de falla fundamental.

Fallas del sitio del coordinador

Son posibles los siguientes casos:

1. El coordinador falla en el estado INICIAL. Esto ocurre antes de que el coordinador haya iniciado el proceso de confirmación. Por lo tanto, iniciará el proceso de confirmación.
al recuperarse.
2. El coordinador falla mientras está en estado de ESPERA. En este caso, el coordinador envió el comando "preparar". Tras la recuperación, el coordinador reiniciará el...

Confirme el proceso para esta transacción desde el principio enviando el mensaje "preparar" una vez más.

3. El coordinador falla mientras se encuentra en los estados COMMIT o ABORT. En este caso, el coordinador habrá informado a los participantes de su decisión y finalizado la transacción. Por lo tanto, al recuperarse, no necesita hacer nada si se han recibido todos los acuses de recibo. De lo contrario, se activa el protocolo de terminación.

Fallas en el sitio del participante

Hay tres alternativas a considerar:

1. Un participante falla en el estado INICIAL. Tras la recuperación, el participante debe abortar la transacción unilateralmente. Veamos por qué esto es aceptable. Tenga en cuenta que el coordinador estará en el estado INICIAL o ESPERA con respecto a esta transacción. Si está en el estado INICIAL, enviará un mensaje de "preparación" y luego pasará al estado ESPERA. Debido al fallo del sitio del participante, no recibirá la decisión del participante y se agotará el tiempo de espera en ese estado. Ya hemos explicado cómo el coordinador gestionaría los tiempos de espera en el estado ESPERA abortando globalmente la transacción.
2. Un participante falla mientras se encuentra en estado LISTO. En este caso, el coordinador ha sido informado de la decisión afirmativa del sitio fallido sobre la transacción antes del fallo. Tras la recuperación, el participante del sitio fallido puede considerar esto como un tiempo de espera en estado LISTO y transferir la transacción incompleta a su protocolo de terminación.
3. Un participante falla mientras se encuentra en el estado ABORT o COMMIT. Estos estados representan las condiciones de terminación, por lo que, al recuperarse, el participante no necesita realizar ninguna acción especial.

Casos adicionales

Consideremos ahora los casos que pueden surgir al relajar el supuesto de atomicidad de las acciones de registro y envío de mensajes. En particular, suponemos que un fallo del sitio puede ocurrir después de que el coordinador o un participante haya escrito un registro, pero antes de que pueda enviar un mensaje. Para este análisis, el lector puede consultar la figura 5.11.

1. El coordinador falla después de escribir el registro begin_commit en el registro, pero antes de enviar el comando "prepare". El coordinador reaccionaría como un fallo en el estado de espera (caso 2 de los fallos del coordinador mencionados anteriormente) y enviaría el comando "prepare" al recuperarse.
2. El sitio de un participante falla después de escribir el registro de lista en el registro, pero antes de enviar el mensaje de confirmación de voto. El participante fallido considera esto como el caso 2 de los fallos de participantes mencionados anteriormente.

3. Un sitio participante falla después de escribir el registro de cancelación en el registro, pero antes de enviar el mensaje de "voto-aborto". Esta es la única situación que no se contempla en los casos fundamentales descritos anteriormente. Sin embargo, en este caso, el participante no necesita hacer nada al recuperarse. El coordinador se encuentra en estado de espera y se agota el tiempo de espera. El protocolo de terminación del coordinador para este estado cancela la transacción globalmente.
4. El coordinador falla tras registrar su decisión final (Abortar o Confirmar), pero antes de enviar su mensaje de "abortar global" o "confirmar global" a los participantes. El coordinador lo considera como su caso 3, mientras que los participantes lo consideran un tiempo de espera en el estado LISTO.
5. Un participante falla después de registrar una cancelación o confirmación , pero antes de enviar el mensaje de confirmación al coordinador. El participante puede tratar esto como su caso 3. El coordinador lo gestionará mediante un tiempo de espera en el estado de confirmación o cancelación.

5.4.3.2 Protocolo de confirmación de tres fases

Como se mencionó anteriormente, los protocolos de confirmación bloqueantes no son recomendables. El protocolo de confirmación en tres fases (3PC) está diseñado como un protocolo no bloqueante cuando los fallos se limitan a fallos del sitio. Cuando se producen fallos de red, la situación se complica.

El algoritmo 3PC es interesante desde un punto de vista algorítmico, pero genera una alta sobrecarga de comunicación en términos de latencia, ya que implica tres rondas de mensajes con escrituras forzadas en el registro estable. Por lo tanto, no se ha adoptado en sistemas reales; incluso el algoritmo 2PC recibe críticas por su alta latencia debido a las fases secuenciales con escrituras forzadas en el registro. Por lo tanto, resumimos el enfoque sin entrar en un análisis detallado.

Consideremos primero las condiciones necesarias y suficientes para diseñar protocolos de compromiso atómico no bloqueantes. Un protocolo de compromiso síncrono dentro de una transición de estado es no bloqueante si, y solo si, su diagrama de transición de estado no contiene ninguno de los siguientes:

1. Ningún estado que sea "adyacente" a un estado de confirmación y a un estado de aborto.
2. No existe ningún estado no confirmable que sea "adyacente" a un estado de confirmación.

El término adyacente aquí significa que es posible pasar de un estado a otro.
con una única transición de estado.

Considere el estado COMMIT en el protocolo 2PC (véase la Fig. 5.10). Si algún proceso se encuentra en este estado, sabemos que todos los sitios han votado para confirmar la transacción.

Estos estados se denominan "comprometibles". Existen otros estados en el protocolo 2PC que no son "comprometibles". El que nos interesa es el estado "LISTO", que no es "comprometible", ya que la existencia de un proceso en este estado no implica que todos los procesos hayan votado para confirmar la transacción.

Es obvio que el estado WAIT en el coordinador y el estado READY en el protocolo 2PC del participante violan las condiciones de no bloqueo que hemos establecido anteriormente.

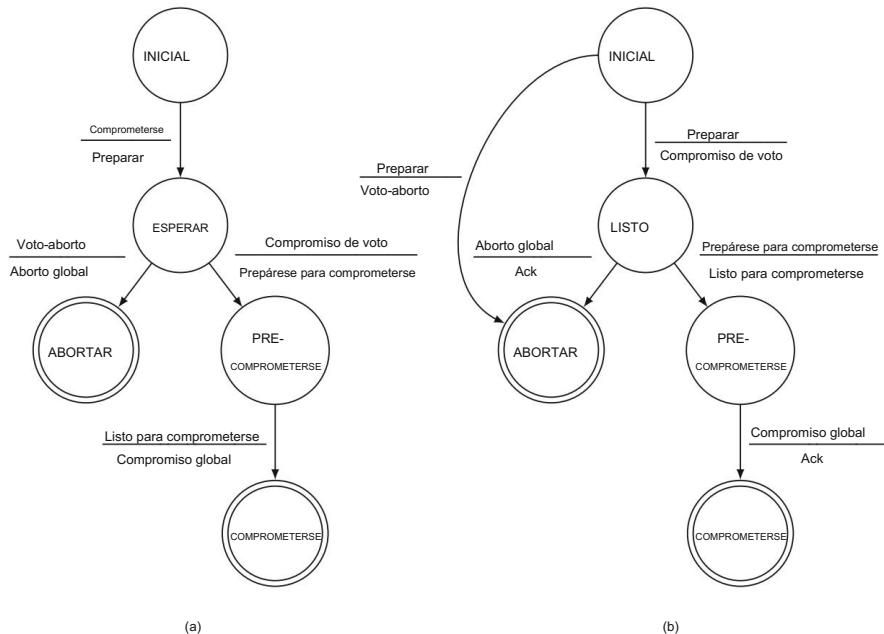


Fig. 5.15 Transiciones de estado en el protocolo 3PC. (a) Estados coordinadores. (b) Estados participantes.

Por lo tanto, se podría realizar la siguiente modificación al protocolo 2PC para satisfacer las condiciones y convertirlo en un protocolo no bloqueante.

Podemos añadir otro estado entre los estados de ESPERA (y LISTO) y COMPROMISO, que actúa como un estado intermedio donde el proceso está listo para comprometerse (si esa es la decisión final), pero aún no lo ha hecho. Los diagramas de transición de estados del coordinador y del participante en este protocolo se muestran en la Fig. 5.15.

Esto se denomina protocolo de confirmación de tres fases (3PC) porque existen tres transiciones de estado, del estado inicial al estado de confirmación. La ejecución del protocolo entre el coordinador y un participante se muestra en la figura 5.16. Cabe destacar que es idéntico a la figura 5.11, salvo por la adición del estado de confirmación previa.

Tenga en cuenta que 3PC también es un protocolo donde todos los estados son síncronos dentro de una transición de estado. Por lo tanto, las condiciones anteriores para 2PC sin bloqueo se aplican a 3PC.

5.4.4 Particionado de red

En esta sección, analizamos cómo los protocolos de confirmaciónatómica que analizamos en la sección anterior pueden gestionar las particiones de red. Las particiones de red se deben a fallos en la línea de comunicación y pueden provocar la pérdida de mensajes, según...

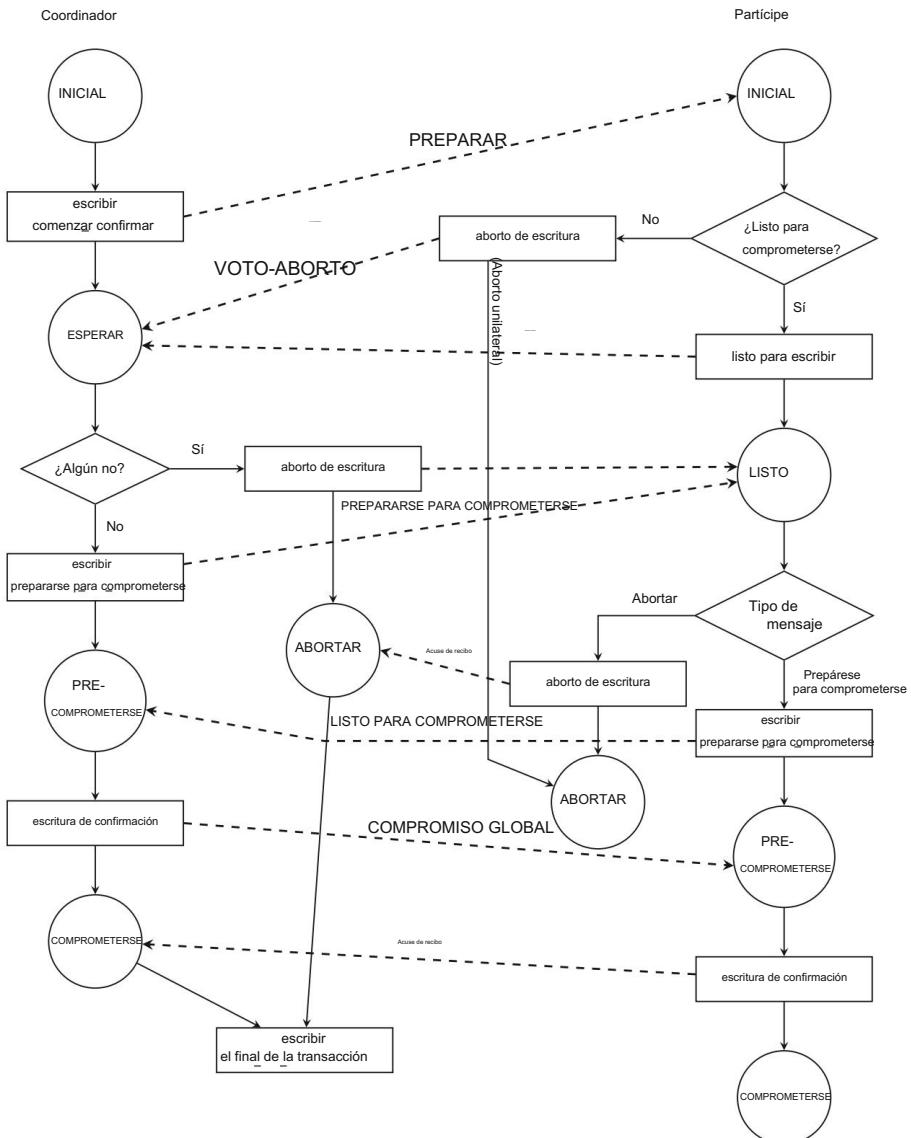


Fig. 5.16 Acciones del protocolo 3PC

Sobre la implementación de la red de comunicación. Una partición se denomina partición simple si la red se divide en solo dos componentes; de lo contrario, se denomina partición múltiple.

Los protocolos de terminación para la partición de red abordan la terminación de las transacciones que estaban activas en cada partición en el momento de la partición. Si se puede...

Al desarrollar protocolos sin bloqueo para terminar estas transacciones, es posible que los sitios de cada partición tomen una decisión de terminación (para una transacción dada) consistente con los sitios de las demás particiones. Esto implicaría que los sitios de cada partición podrían continuar ejecutando transacciones a pesar de la partición.

Desafortunadamente, generalmente no es posible encontrar protocolos de terminación no bloqueantes en presencia de particionamiento de red. Recuerde que nuestras expectativas con respecto a la confiabilidad de la red de comunicación son mínimas. Si un mensaje no se puede entregar, simplemente se pierde. En este caso, se puede demostrar que no existe ningún protocolo de compromiso atómico no bloqueante que sea resistente al particionamiento de red.

Este resultado es bastante negativo, ya que también significa que, si se produce una partición de red, no es posible continuar con las operaciones normales en todas las particiones, lo que limita la disponibilidad de todo el sistema de base de datos distribuida. Sin embargo, un resultado positivo indica que es posible diseñar protocolos de confirmación atómica sin bloqueo que sean resistentes a particiones simples. Desafortunadamente, si se producen múltiples particiones, tampoco es posible diseñar dichos protocolos.

En el resto de esta sección, analizamos varios protocolos que abordan la partición de red en bases de datos no replicadas. El problema es bastante diferente en el caso de las bases de datos replicadas, que abordaremos en el siguiente capítulo.

Ante la partición de red de bases de datos no replicadas, la principal preocupación reside en la finalización de las transacciones activas en el momento de la partición. Cualquier nueva transacción que acceda a un dato almacenado en otra partición simplemente se bloquea y debe esperar a que la red se repare.

Los accesos concurrentes a los elementos de datos dentro de una partición pueden gestionarse mediante el algoritmo de control de concurrencia. Por lo tanto, el problema principal es garantizar que la transacción finalice correctamente. En resumen, el problema de la partición de la red se gestiona mediante el protocolo de confirmación y, más específicamente, mediante los protocolos de terminación y recuperación.

La ausencia de protocolos no bloqueantes que garanticen la asignación atómica de transacciones distribuidas plantea una importante decisión de diseño. Podemos permitir que todas las particiones continúen sus operaciones normales y aceptar que la consistencia de la base de datos pueda verse comprometida, o garantizar la consistencia de la base de datos empleando estrategias que permitan la operación en una de las particiones mientras los sitios en las demás permanecen bloqueados. Este problema de decisión es la premisa de una clasificación de estrategias de gestión de particiones. Las estrategias pueden clasificarse como pesimistas u optimistas. Las estrategias pesimistas enfatizan la consistencia de la base de datos y, por lo tanto, no permitirían la ejecución de transacciones en una partición si no se garantiza que se mantenga dicha consistencia.

Los enfoques optimistas, por otro lado, enfatizan la disponibilidad de la base de datos incluso si esto causara inconsistencias.

La segunda dimensión se relaciona con el criterio de corrección. Si se utiliza la serializabilidad como criterio fundamental de corrección, estas estrategias se denominan sintácticas, ya que la teoría de la serializabilidad utiliza únicamente información sintáctica. Sin embargo, si se utiliza un criterio de corrección más abstracto, que depende de la semántica de las transacciones o de la base de datos, las estrategias se denominan semánticas.

De acuerdo con el criterio de corrección adoptado en este libro (serializabilidad), en esta sección solo consideramos enfoques sintácticos. Las dos secciones siguientes describen diversas estrategias sintácticas para bases de datos no replicadas.

Todos los protocolos de terminación conocidos que gestionan la partición de red en el caso de bases de datos no replicadas son pesimistas. Dado que los enfoques pesimistas priorizan el mantenimiento de la consistencia de la base de datos, la cuestión fundamental que debemos abordar es cuál de las particiones puede continuar operando con normalidad. Consideraremos dos enfoques.

5.4.4.1 Protocolos centralizados

Los protocolos de terminación centralizada se basan en los algoritmos de control de concurrencia centralizados descritos en la sección 5.2. En este caso, conviene permitir el funcionamiento de la partición que contiene el sitio central, ya que esta gestiona las tablas de bloqueo.

Las técnicas de sitio primario están centralizadas respecto a cada elemento de datos. En este caso, puede haber más de una partición operativa para diferentes consultas. Para cualquier consulta, solo la partición que contiene el sitio primario de los elementos de datos del conjunto de escritura de esa transacción puede ejecutarla.

Ambos enfoques son sencillos y funcionarían bien, pero dependen de un mecanismo específico de control de concurrencia. Además, se espera que cada sitio pueda diferenciar correctamente la partición de la red de los fallos del sitio. Esto es necesario, ya que los participantes en la ejecución del protocolo de confirmación reaccionan de forma distinta a los distintos tipos de fallos.

Desafortunadamente, esto generalmente no es posible.

5.4.4.2 Protocolos basados en votación

La votación también puede utilizarse para gestionar accesos concurrentes a datos. Se ha propuesto una votación directa por mayoría como método de control de concurrencia para bases de datos completamente replicadas. La idea fundamental es que una transacción se ejecuta si la mayoría de los sitios votan a favor de ejecutarla.

La idea de la votación por mayoría se ha generalizado a la votación con quórum. La votación basada en quórum puede utilizarse como método de control de réplicas (como se explica en el siguiente capítulo), así como como método de confirmación para garantizar la atomicidad de las transacciones en presencia de particiones de red. En el caso de bases de datos no replicadas, esto implica la integración del principio de votación con los protocolos de confirmación. Presentamos una propuesta específica en este sentido.

A cada sitio del sistema se le asigna un voto V_i . Supongamos que el número total y los quórum El número de votos en el sistema V , de cancelación y confirmación son V_a y V_c .
es V , respectivamente. Por lo tanto, se deben cumplir las siguientes reglas al implementar el protocolo de confirmación:

1. $V_a + V_c > V$, donde $0 \leq V_a, V_c \leq V$.

2. Antes de que una transacción se confirme, debe obtener un quórum de compromiso Vc.
3. Antes de que una transacción se aborte, debe obtener un quórum de aborto Va.

La primera regla garantiza que una transacción no pueda confirmarse y cancelarse simultáneamente.

Las dos siguientes indican los votos que debe obtener una transacción antes de que pueda terminar de una forma u otra. La integración de técnicas de quórum en los protocolos de confirmación se deja como ejercicio.

5.4.5 Protocolo de Consenso de Paxos

Hasta este punto, hemos estudiado los protocolos 2PC para alcanzar un acuerdo entre los gestores de transacciones respecto a la resolución de una transacción distribuida y hemos descubierto que presentan la propiedad indeseable de bloquearse cuando el coordinador y otro participante fallan. Analizamos cómo solucionar esto mediante el protocolo 3PC, que es costoso y no es resistente a la partición de la red. Nuestro enfoque en la partición de la red consideró la votación para determinar la partición donde reside la mayoría de los gestores de transacciones y finalizar la transacción en dicha partición. Estas pueden parecer soluciones parciales al problema fundamental de encontrar mecanismos tolerantes a fallos para alcanzar un acuerdo (consenso) entre los gestores de transacciones sobre el destino de la transacción en cuestión. Resulta que alcanzar un consenso entre sitios es un problema general en la computación distribuida, conocido como consenso distribuido. Se han propuesto diversos algoritmos para abordar este problema; en esta sección, analizamos la familia de algoritmos Paxos y mencionamos otros en las Notas Bibliográficas.

Primero analizaremos Paxos en el contexto general en el que se definió originalmente y luego consideraremos cómo se puede usar en protocolos de confirmación. En el contexto general, el algoritmo logra un consenso entre los sitios sobre el valor de una variable (o decisión). Es importante considerar que se alcanza un consenso si la mayoría de los sitios están de acuerdo con el valor, no todos. Por lo tanto, un cierto número de sitios puede fallar, pero mientras exista una mayoría, se puede alcanzar el consenso. Se identifican tres roles: el proponente , que recomienda un valor para la variable; el aceptante , que decide si acepta o no el valor recomendado; y el aprendiz , que descubre el valor acordado preguntando a uno de los aprendices (el valor es impulsado por un aceptante). Cabe destacar que estos roles pueden ubicarse en un sitio, pero cada sitio solo puede tener una instancia de cada uno. Los aprendices no son muy importantes, por lo que no los consideraremos en detalle en nuestra exposición.

El protocolo Paxos es sencillo si solo hay un proponente y funciona como el protocolo 2PC: en la primera ronda, el proponente sugiere un valor para la variable y los aceptantes envían sus respuestas (aceptar/no aceptar). Si el proponente obtiene la aceptación de la mayoría de los aceptantes, determina ese valor como el valor de la variable y notifica a los aceptantes, quienes lo registran como el valor final. Un aprendiz puede, en cualquier momento, preguntar a un aceptante cuál es el valor de la variable y obtener el valor más reciente.

Por supuesto, la realidad no es tan simple y el protocolo Paxos debe poder...
afrontar las siguientes complicaciones:

1. Dado que este es, por definición, un protocolo de consenso distribuido, varios proponentes pueden proponer un valor para la misma variable. Por lo tanto, quien acepta debe elegir uno de los valores propuestos.
2. Dadas múltiples propuestas, es posible obtener votos divididos sobre varias propuestas. sin que ninguna propuesta de valor obtuviera la mayoría.
3. Es posible que algunos de los aceptantes fracasen tras aceptar un valor. Si los aceptantes restantes que aceptaron ese valor no constituyen una mayoría, esto genera un problema.

Paxos soluciona el primer problema mediante un número de votación para que los aceptantes puedan diferenciar las diferentes propuestas, como se explica más adelante. El segundo problema puede solucionarse ejecutando múltiples rondas de consenso: si ninguna propuesta alcanza la mayoría, se ejecuta otra ronda y se repite hasta que un valor la alcance. En algunos casos, esto puede prolongarse durante varias iteraciones, lo que puede reducir su rendimiento. Paxos soluciona el problema designando un líder al que cada proponente envía su propuesta de valor. El líder elige un valor para cada variable y busca obtener la mayoría. Esto reduce la naturaleza distribuida del protocolo de consenso. En diferentes rondas de la ejecución del protocolo, el líder podría ser diferente. El tercer problema es más grave. De nuevo, esto podría tratarse como el segundo problema y se podría iniciar una nueva ronda. Sin embargo, la complicación radica en que algunos participantes pueden haber aprendido el valor aceptado de los aceptantes en la ronda anterior, y si se elige un valor diferente en la nueva ronda, se produce una inconsistencia. Paxos soluciona esto nuevamente mediante el uso de números de votación.

Presentamos a continuación los pasos del "Paxos básico" que se centra en determinar el valor de una única variable (de ahí la omisión del nombre de la variable en lo que sigue). Paxos básico también simplifica la determinación de los números de votación: en este caso, el número de votación solo debe ser único y monótono para cada proponente; no hay ningún intento de hacerlos globalmente únicos ya que se alcanza un consenso cuando la mayoría de los aceptantes se ponen de acuerdo sobre algún valor para la variable, independientemente de quién lo propuso.

A continuación se muestra el funcionamiento básico de Paxos en ausencia de fallos:

- S1. El proponente que desea iniciar un consenso envía a todos los aceptantes un mensaje de «preparación» con su número de papeleta [prepare(bal)].
- S2. Cada aceptante que recibe el mensaje de preparación realiza lo siguiente: si no ha recibido ninguna propuesta antes, registra prepare(bal) en su registro y responde con ack(bal); de lo contrario, si bal > cualquier número de votación que haya recibido de cualquier proponente antes, registra prepare(bal) en su registro y responde con el número de votación (bal) y el valor (val) de la propuesta con el número más alto que haya aceptado antes de esto: ack(bal, bal); de lo contrario, ignora el mensaje de preparación. , val);

S3. Cuando el proponente recibe un mensaje de acuse de recibo de un aceptante, lo registra.
mensaje.

S4. Cuando el proponente ha recibido acuses de recibo de la mayoría de los aceptantes (es decir, tiene quórum para establecer un consenso), envía un mensaje accept(nbal, val) a los aceptantes, donde nbal es el número de papeleta a aceptar y val es el valor a aceptar. Donde nbal y val se determinan de la siguiente manera: si todos los mensajes de acuse de recibo que recibió indican que ningún aceptante ha aceptado previamente un valor

entonces, el valor propuesto val se establece en lo que el proponente quería sugerir en primer lugar y $nbal \leftarrow bal$; de lo contrario, val se establece en el val en los mensajes de confirmación de retorno con el bal más grande y $nbal \leftarrow bal$ (de modo que todos convergen al valor con el número de votación más grande);

El proponente ahora envía accept(bal, val) a todos los aceptantes (val es el valor propuesto).

S5. Cada aceptador realiza lo siguiente al recibir la aceptación (nbal, val):

mensaje:
si $nbal = ack.bal$ (es decir, el número de balota del mensaje de aceptación es el que había prometido anteriormente),
entonces registra aceptado(nbal, val); de lo contrario, ignora el mensaje.

Varios puntos sobre el protocolo anterior. Primero, cabe destacar que en el paso S2, un aceptante ignora el mensaje de preparación si ya ha recibido un mensaje de preparación superior al que acaba de recibir. Aunque el protocolo funciona correctamente en este caso, el proponente puede continuar comunicándose con este aceptante más adelante (p. ej., en el paso S5), lo cual se puede evitar enviando un acuse de recibo negativo para que el proponente pueda descartarlo. El segundo punto es que cuando un aceptante acusa recibo de un mensaje de preparación, también reconoce que es el líder de Paxos para esta ronda. Por lo tanto, en cierto sentido, la selección del líder forma parte del protocolo. Sin embargo, para solucionar el segundo problema mencionado anteriormente, es posible designar un líder, que inicia las rondas. Si esta es la implementación elegida, el líder elegido puede elegir la propuesta que presentará si recibe varias propuestas. Finalmente, dado que el protocolo avanza si hay una mayoría de participantes (sitios) disponibles, para un sistema con N sitios, puede tolerar $N - 1$ fallas simultáneas del sitio. 2

Analicemos brevemente cómo Paxos gestiona los fallos. El caso más sencillo es cuando algunos aceptantes fallan, pero aún existe el quórum necesario para tomar una decisión. En este caso, el protocolo continúa como de costumbre. Si suficientes aceptantes no logran eliminar la posibilidad de quórum, el protocolo lo gestiona de forma natural mediante una nueva votación (o varias) cuando se logre. El caso que siempre supone un reto para los algoritmos de consenso es el fallo del proponente (que también es el líder); en este caso, Paxos elige un nuevo líder mediante algún mecanismo (existen diversas propuestas en la literatura para esto) y este inicia una nueva ronda con un nuevo número de votación.

Paxos cuenta con la característica de toma de decisiones multironda de 2PC y 3PC, así como con el método de votación mayoritaria de los algoritmos de quórum; los generaliza en un protocolo coherente para alcanzar el consenso en presencia de un conjunto de procesos distribuidos. Se ha señalado que 2PC y 3PC (y otros protocolos de confirmación) son casos especiales de Paxos. A continuación, describimos una propuesta para ejecutar 2PC con Paxos y lograr un protocolo 2PC no bloqueante, denominado Paxos 2PC.

En Paxos 2PC, los gestores de transacciones actúan como líderes; esto significa que lo que antes llamábamos coordinador ahora se denomina líder. Una característica principal del protocolo es que el líder utiliza un protocolo de Paxos para alcanzar el consenso y registrar su decisión en un registro replicado. La primera característica es importante, ya que el protocolo no necesita que todos los participantes estén activos y participen en la decisión; basta con una mayoría y los demás pueden converger en el valor decidido cuando se recuperen. La segunda es importante porque los fallos del líder (coordinador) ya no son un obstáculo: si un líder falla, se puede elegir un nuevo líder y el estado de la decisión de la transacción está disponible en el registro replicado en otros sitios.

5.4.6 Consideraciones arquitectónicas

En las secciones anteriores, analizamos los protocolos de confirmación atómica a nivel abstracto. Veamos ahora cómo implementarlos en el marco de nuestro modelo arquitectónico. Esta discusión implica la especificación de la interfaz entre los algoritmos de control de concurrencia y los protocolos de confiabilidad.

En ese sentido, las discusiones de este capítulo se relacionan con la ejecución de los comandos Commit, Abort y recovery .

No es sencillo especificar con precisión la ejecución de estos comandos por dos razones. En primer lugar, para su correcta implementación, se requiere un modelo de arquitectura significativamente más detallado que el presentado. En segundo lugar, el esquema general de implementación depende en gran medida de los procedimientos de recuperación que implementa el gestor de recuperación local. Por ejemplo, la implementación del protocolo 2PC sobre un LRM que emplea un esquema de recuperación sin corrección ni vaciado es muy diferente a su implementación sobre un LRM que emplea un esquema de recuperación con corrección y vaciado. Las alternativas son simplemente demasiadas. Por lo tanto, limitamos nuestro análisis arquitectónico a tres áreas: la implementación de los conceptos de coordinador y participante para los protocolos de control de confirmación y réplica en el marco de la arquitectura del gestor de transacciones, programador y gestor de recuperación local; el acceso del coordinador al registro de la base de datos; y los cambios necesarios en las operaciones del gestor de recuperación local.

Una posible implementación de los protocolos de confirmación dentro de nuestro modelo arquitectónico es ejecutar los algoritmos del coordinador y del participante dentro de los gestores de transacciones en cada sitio. Esto proporciona cierta uniformidad en la ejecución de las operaciones de confirmación distribuidas. Sin embargo, implica una comunicación innecesaria entre el gestor de transacciones del participante y su programador; esto se debe a que

El programador debe decidir si una transacción puede confirmarse o cancelarse.

Por lo tanto, puede ser preferible implementar el coordinador como parte del administrador de transacciones y al participante como parte del programador. Si el programador implementa un algoritmo de control de concurrencia estricto (es decir, no permite abortos en cascada), estará listo automáticamente para confirmar la transacción cuando llegue el mensaje de preparación. La prueba de esta afirmación se deja como ejercicio. Sin embargo, incluso esta alternativa de implementar el coordinador y el participante fuera del procesador de datos tiene problemas. El primer problema es la administración del registro de la base de datos. Recuerde que el registro de la base de datos es mantenido por el LRM y el administrador de búfer. Sin embargo, la implementación del protocolo de confirmación como se describe aquí requiere que el administrador de transacciones y el programador también accedan al registro. Una posible solución a este problema es mantener un registro de confirmación (que podría llamarse registro de transacciones distribuidas) al que acceda el administrador de transacciones y que esté separado del registro de la base de datos que mantienen el LRM y el administrador de búfer. La otra alternativa es escribir los registros del protocolo de confirmación en el mismo registro de la base de datos. Esta segunda alternativa tiene varias ventajas. Primero, solo se mantiene un registro; Esto simplifica los algoritmos necesarios para guardar los registros en un almacenamiento estable. Más importante aún, la recuperación ante fallos en una base de datos distribuida requiere la cooperación del administrador de recuperación local y el programador (es decir, el participante). Un único registro de la base de datos puede servir como repositorio central de la información de recuperación para ambos componentes.

Un segundo problema asociado con la implementación del coordinador dentro del gestor de transacciones y del participante como parte del planificador es la integración con los protocolos de control de concurrencia. Esta implementación se basa en que los planificadores determinen si una transacción puede confirmarse. Esto es adecuado para algoritmos de control de concurrencia distribuidos donde cada sitio cuenta con un planificador.

Sin embargo, en protocolos centralizados como el 2PL centralizado, solo existe un planificador en el sistema. En este caso, los participantes pueden implementarse como parte de los procesadores de datos (más precisamente, como parte de los gestores de recuperación locales), lo que requiere modificaciones tanto en los algoritmos implementados por el LRM como, posiblemente, en la ejecución del protocolo 2PC. Dejaremos los detalles para ejercicios posteriores.

El almacenamiento de los registros del protocolo de confirmación en el registro de la base de datos que mantiene el LRM y el gestor de búfer requiere algunos cambios en los algoritmos del LRM. Este es el tercer problema arquitectónico que abordamos. Estos cambios dependen del tipo de algoritmo que utiliza el LRM. En general, los algoritmos del LRM deben modificarse para gestionar por separado el comando de preparación y las decisiones de confirmación global (o cancelación global). Además, tras la recuperación, el LRM debe modificarse para leer el registro de la base de datos e informar al planificador sobre el estado de cada transacción, de modo que se puedan seguir los procedimientos de recuperación descritos anteriormente. Analicemos con más detalle esta función del LRM.

El LRM primero debe determinar si el sitio fallido es el host del coordinador o de un participante. Esta información puede almacenarse junto con el registro `Begin_transaction`. A continuación, el LRM debe buscar el último registro escrito.

en el registro durante la ejecución del protocolo de confirmación. Si ni siquiera encuentra un registro begin_commit (en el sitio del coordinador) ni un registro de cancelación o confirmación (en los sitios participantes), la transacción no ha comenzado a confirmarse. En este caso, el LRM puede continuar con su proceso de recuperación. Sin embargo, si el proceso de confirmación ya ha comenzado, la recuperación debe transferirse al coordinador. Por lo tanto, el LRM envía el último registro al planificador.

5.5 Enfoques modernos para escalar la gestión de transacciones

Todos los algoritmos presentados anteriormente introducen cuellos de botella en diferentes puntos del procesamiento transaccional. Aquellos que implementan serialización limitan considerablemente la concurrencia potencial debido a conflictos entre consultas extensas que leen muchos datos y actualizan transacciones. Por ejemplo, una consulta analítica que realiza un escaneo completo de la tabla con un predicado que no se basa en la clave principal causa un conflicto con cualquier actualización de la tabla. Todos los algoritmos requieren un paso de procesamiento centralizado para confirmar las transacciones una por una.

Esto crea un cuello de botella, ya que no pueden procesar transacciones a una velocidad superior a la de un solo nodo. Los algoritmos de bloqueo requieren la gestión de interbloqueos y muchos utilizan la detección de interbloqueos, lo cual resulta difícil en un entorno distribuido, como ya comentamos. El algoritmo que presentamos en la sección anterior sobre el aislamiento de instantáneas realiza una certificación centralizada, lo que nuevamente genera un cuello de botella.

Escalar la ejecución de transacciones para lograr un alto rendimiento en sistemas distribuidos o paralelos ha sido un tema de interés durante mucho tiempo. En los últimos años, han surgido soluciones; en esta sección, analizamos dos enfoques: Google Spanner y LeanXcale. Ambos implementan cada una de las propiedades ACID de forma escalable y componible. En ambos enfoques, se propone una nueva técnica para serializar transacciones que pueden soportar tasas de rendimiento muy altas (millones o incluso miles de millones de transacciones por segundo). Ambos enfoques permiten registrar la confirmación de las transacciones y utilizar esta marca de tiempo para serializarlas.

Spanner utiliza tiempo real para registrar las transacciones, mientras que LeanXcale utiliza tiempo lógico. El tiempo real tiene la ventaja de no requerir comunicación, pero sí alta precisión y una infraestructura de tiempo real altamente confiable. La idea es usar el tiempo real como marca de tiempo y esperar a que transcurra la precisión para que el resultado sea visible para las transacciones.

LeanXcale adopta un enfoque en el que las transacciones se marcan con una hora lógica y las transacciones confirmadas se hacen visibles progresivamente a medida que las brechas en el orden de serialización se completan con nuevas transacciones confirmadas. La hora lógica evita tener que depender de la creación de una infraestructura en tiempo real.

5.5.1 Llave inglesa

Spanner utiliza bloqueo tradicional y 2PC, y proporciona serialización como nivel de aislamiento. Dado que el bloqueo genera una alta contención entre consultas grandes y transacciones de actualización, Spanner también implementa multiversionado. Para evitar el cuello de botella de la certificación centralizada, a los datos actualizados se les asignan marcas de tiempo (en tiempo real) al confirmarse. Para ello, Spanner implementa un servicio interno llamado TrueTime que proporciona la hora actual y su precisión. Para que el servicio TrueTime sea fiable y preciso, utiliza relojes atómicos y GPS, ya que tienen diferentes modos de fallo que pueden compensarse entre sí.

Por ejemplo, los relojes atómicos presentan una deriva continua, mientras que el GPS pierde precisión en ciertas condiciones meteorológicas, cuando la antena se rompe, etc. La hora actual obtenida mediante TrueTime se utiliza para marcar las transacciones que se van a confirmar. La precisión reportada se utiliza para compensar durante la asignación de la marca de tiempo: tras obtener la hora local, hay un tiempo de espera equivalente a la duración de la inexactitud, normalmente de unos 10 milisegundos. Para solucionar los bloqueos, Spanner adopta la prevención de bloqueos mediante un enfoque de "wound-and-wait" (véase el Apéndice C), eliminando así el cuello de botella que supone la detección de bloqueos.

La gestión del almacenamiento en Spanner se hace escalable al aprovechar Google Bigtable, un almacén de datos de clave-valor (véase Cap. 11).

El multiversionado se implementa de la siguiente manera. Las versiones privadas de los datos se conservan en cada sitio hasta su confirmación. Tras la confirmación, se inicia el protocolo 2PC, durante el cual las escrituras almacenadas en el búfer se propagan a cada participante. Cada participante establece bloqueos en los datos actualizados. Una vez adquiridos todos los bloqueos, asigna una marca de tiempo de confirmación mayor que cualquier marca de tiempo asignada previamente. El coordinador también adquiere los bloqueos de escritura. Tras adquirir todos los bloqueos de escritura y recibir el mensaje preparado de todos los participantes, el coordinador elige una marca de tiempo mayor que la hora actual más la inexactitud, y mayor que cualquier otra marca de tiempo asignada localmente. El coordinador espera a que se complete la marca de tiempo asignada (espera de recuperación debido a la inexactitud) y luego comunica la decisión de confirmación al cliente.

Al utilizar multiversiones, Spanner también implementa transacciones de solo lectura que leen sobre la instantánea en el momento actual.

5.5.2 LeanXcale

LeanXcale utiliza un enfoque totalmente diferente para la gestión transaccional escalable. En primer lugar, utiliza tiempo lógico para registrar las transacciones y establecer la visibilidad de los datos confirmados. En segundo lugar, proporciona aislamiento de instantáneas. En tercer lugar, todas las funciones que consumen muchos recursos, como el control de concurrencia, el registro, el almacenamiento y el procesamiento de consultas, son totalmente distribuidas y paralelas, sin coordinación.

Para el tiempo lógico, LeanXcale utiliza dos servicios: el secuenciador de confirmaciones y el servidor de instantáneas. El secuenciador de confirmaciones distribuye las marcas de tiempo de las confirmaciones y el

El servidor de instantáneas regula la visibilidad de los datos confirmados al avanzar la instantánea visible para las transacciones.

Dado que LeanXcale proporciona aislamiento de instantáneas, solo necesita detectar conflictos de escritura. Implementa un gestor de conflictos (posiblemente distribuido). Cada gestor de conflictos se encarga de comprobar los conflictos en un subconjunto de elementos de datos. Básicamente, un gestor de conflictos recibe solicitudes de los LTM para comprobar si un elemento de datos que se va a actualizar entra en conflicto con alguna actualización simultánea. Si existe un conflicto, el LTM abortará la transacción; de lo contrario, podrá continuar sin abortar.

La funcionalidad de almacenamiento la proporciona un almacén de datos relacional de clave-valor llamado KiVi. Las tablas se dividen horizontalmente en unidades llamadas regiones. Cada región se almacena en un servidor KiVi.

Cuando se inicia una confirmación, un LTM la gestiona y comienza el procesamiento de la confirmación de la siguiente manera. Primero, el LTM toma una marca de tiempo de confirmación del rango local y la utiliza para marcar el conjunto de escritura de la transacción, que posteriormente se registra. El registro se escala mediante el uso de múltiples registradores. Cada registrador gestiona un subconjunto de LTM. Un registrador responde al LTM cuando el conjunto de escrituras es duradero. LeanXcale implementa multiversionado en la capa de almacenamiento. Al igual que en Spanner, existen copias privadas en cada sitio. Cuando el conjunto de escrituras es duradero, las actualizaciones se propagan a los servidores KiVi correspondientes, con la marca de tiempo de confirmación. Una vez propagadas todas las actualizaciones, la transacción es legible si se utiliza la marca de tiempo de inicio correcta (igual o posterior a la marca de tiempo de confirmación). Sin embargo, sigue siendo invisible. A continuación, el LTM informa al servidor de instantáneas que la transacción es duradera y legible. El servidor de instantáneas registra la instantánea actual, es decir, la marca de tiempo de inicio que utilizarán las nuevas transacciones. También registra las transacciones duraderas y legibles con una marca de tiempo de confirmación posterior a la instantánea actual. Siempre que no haya intervalos entre la instantánea actual y una marca de tiempo, el servidor de instantáneas avanza la instantánea a esa marca de tiempo. En ese momento, los datos confirmados con una marca de tiempo de confirmación inferior a la de la instantánea actual se vuelven visibles para las nuevas transacciones, ya que obtendrán una marca de tiempo de inicio en la instantánea actual. Veamos cómo funciona con un ejemplo.

Ejemplo 5.5 Considere 5 transacciones que se confirman en paralelo con las marcas de tiempo de confirmación del 11 al 15. Sea la instantánea actual en el servidor de instantáneas 10. El orden en que los LTM informan al servidor de instantáneas que las transacciones son duraderas y legibles es 11, 15, 14, 12 y 13. Cuando el servidor de instantáneas recibe una notificación sobre una transacción con la marca de tiempo de confirmación 11, avanza la instantánea del 10 al 11, ya que no hay espacios. Con la transacción con la marca de tiempo de confirmación 15, no puede avanzar la instantánea, ya que de lo contrario las nuevas transacciones podrían observar un estado inconsistente que omite las actualizaciones de las transacciones con las marcas de tiempo de confirmación del 12 al 14. Ahora la transacción con la marca de tiempo 14 informa que sus actualizaciones son duraderas y legibles. Nuevamente, la instantánea no puede avanzar. Pero ahora la transacción con la marca de tiempo de confirmación 13 se vuelve duradera y legible, y ahora la instantánea avanza hasta el 15, ya que no hay espacios en el orden de serialización.

Tenga en cuenta que, si bien el algoritmo hasta ahora proporciona aislamiento de instantáneas, no proporciona consistencia de sesión, una característica deseable para que las transacciones dentro de la misma sesión puedan leer las escrituras de transacciones previamente confirmadas. Para garantizar la consistencia de sesión, se ha añadido un nuevo mecanismo. Básicamente, cuando una sesión confirma una transacción que realizó actualizaciones, esperará a que la instantánea avance más allá de la marca de tiempo de confirmación de la transacción de actualización para iniciarse y, a continuación, comenzará con una instantánea que garantice la observación de sus propias escrituras.

5.6 Conclusión

En este capítulo, analizamos cuestiones relacionadas con el procesamiento distribuido de transacciones. Una transacción es una unidad atómica de ejecución que transforma una base de datos consistente en otra. Las propiedades ACID de las transacciones también indican los requisitos para su gestión. La consistencia requiere una definición de la aplicación de la integridad (que abordamos en el capítulo 3), así como algoritmos de control de concurrencia. El control de concurrencia también aborda el problema del aislamiento. El mecanismo de control de concurrencia distribuida de un SGBD distribuido garantiza el mantenimiento de la consistencia de la base de datos distribuida y, por lo tanto, es uno de los componentes fundamentales de un SGBD distribuido. Introdujimos algoritmos de control de concurrencia distribuida de tres tipos: basados en bloqueos, basados en la ordenación de marcas de tiempo y optimistas. Observamos que los algoritmos basados en bloqueos pueden provocar bloqueos distribuidos (o globales) e introdujimos un enfoque para detectarlos y resolverlos.

Las propiedades de durabilidad y atomicidad de las transacciones requieren un análisis de la fiabilidad de los SGBD distribuidos. En concreto, la durabilidad se sustenta en diversos protocolos de confirmación y gestión de confirmaciones, mientras que la atomicidad requiere el desarrollo de protocolos de recuperación adecuados. Presentamos dos protocolos de confirmación, 2PC y 3PC, que garantizan la atomicidad y la durabilidad de las transacciones distribuidas incluso en caso de fallo. Uno de estos algoritmos (3PC) puede configurarse como no bloqueante, lo que permitiría que cada sitio continuara su funcionamiento sin esperar la recuperación del sitio fallido. El rendimiento de los protocolos de confirmación distribuidos con respecto a la sobrecarga que añaden a los algoritmos de control de concurrencia es un tema interesante.

Lograr un alto rendimiento transaccional en sistemas de gestión de bases de datos (SGBD) distribuidos y paralelos ha sido un tema de interés desde hace tiempo, con avances positivos en los últimos años. Analizamos dos enfoques para lograr este objetivo en el marco de los sistemas Spanner y LeanXcale.

Hay algunas cuestiones que hemos omitido de este capítulo:

1. Modelos de transacción avanzados. El modelo de transacción que utilizamos en este capítulo se conoce comúnmente como modelo de transacción plana, con un único punto de inicio (`Begin_transaction`) y un único punto de finalización (`End_transaction`). La mayor parte del trabajo de gestión de transacciones en bases de datos se ha centrado en transacciones planas. Sin embargo, existen modelos de transacción más avanzados. Uno de ellos se denomina

Modelo de transacción anidada , donde una transacción incluye otras transacciones con sus propios puntos de inicio y fin. Las transacciones integradas en otra suelen denominarse subtransacciones. La estructura de una transacción anidada es un trie donde la transacción más externa es la raíz, y las subtransacciones se representan como los demás nodos. Estas se diferencian en sus características de terminación.

Una categoría, denominada transacciones anidadas cerradas, se compromete de forma ascendente a través de la raíz. Por lo tanto, una subtransacción anidada comienza después de su padre y termina antes , y la confirmación de las subtransacciones depende de la confirmación de la matriz. La semántica de estas transacciones impone atomicidad en el nivel superior. La alternativa es la anidación abierta, que flexibiliza la restricción de atomicidad de nivel superior de las transacciones anidadas cerradas. Por lo tanto, una transacción anidada abierta permite que sus resultados parciales se observen fuera de la transacción. Las sagas y las transacciones divididas son ejemplos de anidación abierta.

Las transacciones aún más avanzadas son los flujos de trabajo. Lamentablemente, el término «flujo de trabajo» no tiene un significado claro y uniforme. Una definición práctica es la de un conjunto de tareas con un orden parcial entre ellas que, en conjunto, realizan un proceso complejo.

Si bien la gestión de estos modelos de transacciones avanzadas es importante, Están fuera de nuestro alcance, por lo que no los hemos considerado en este capítulo.

2. Supuestos sobre las transacciones. En nuestras discusiones, no distinguimos entre transacciones de solo lectura y transacciones de actualización. Es posible mejorar significativamente el rendimiento de las transacciones que solo leen datos, o de sistemas con una alta proporción de transacciones de solo lectura respecto de las de actualización.

Estas cuestiones quedan fuera del alcance de este libro.

También hemos tratado los bloqueos de lectura y escritura de forma idéntica. Es posible diferenciarlos y desarrollar algoritmos de control de concurrencia que permiten la "conversión de bloqueos", mediante la cual las transacciones pueden obtener bloqueos en un modo y luego modificar sus modos de bloqueo según cambien sus requisitos. Normalmente, la conversión se realiza de bloqueos de lectura a bloqueos de escritura.

3. Modelos de ejecución de transacciones. Todos los algoritmos descritos asumen un modelo computacional donde el gestor de transacciones, en el sitio de origen de una transacción, coordina la ejecución de cada operación de la base de datos correspondiente. Esto se denomina ejecución centralizada. También es posible considerar un modelo de ejecución distribuida donde una transacción se descompone en un conjunto de subtransacciones, cada una asignada a un sitio donde el gestor de transacciones coordina su ejecución. Esto resulta intuitivamente más atractivo, ya que permite el equilibrio de carga entre los múltiples sitios de una base de datos distribuida.

Sin embargo, los estudios de rendimiento indican que la computación distribuida solo funciona mejor bajo cargas ligeras.

4. Tipos de error. Solo consideramos los fallos atribuibles a errores. En otras palabras, asumimos que se hizo todo lo posible para diseñar e implementar los sistemas (hardware y software), pero que, debido a diversas fallas en los componentes, el diseño o el entorno operativo, estos no funcionaron correctamente. Estos fallos se denominan fallos de omisión. Existe otra clase de fallos, llamados fallos de comisión, en los que los sistemas pueden no haber sido...

Diseñados e implementados para su correcto funcionamiento. La diferencia radica en que, al ejecutar el protocolo 2PC, por ejemplo, si un participante recibe un mensaje del coordinador, este se considera correcto: el coordinador está operativo y envía al participante un mensaje correcto para que lo procese. El único fallo del que debe preocuparse el participante es si el coordinador falla o si sus mensajes se pierden. Estos son fallos de omisión. Si, por otro lado, los mensajes que recibe un participante no son confiables, este también debe lidiar con fallos de comisión. Por ejemplo, un sitio participante puede hacerse pasar por el coordinador y enviar un mensaje malicioso. No hemos analizado las medidas de confiabilidad necesarias para abordar este tipo de fallos. Las técnicas que abordan los fallos de comisión se denominan típicamente acuerdo bizantino.

Además de estos temas, existe una gran cantidad de trabajos recientes sobre la gestión de transacciones en diversos entornos (p. ej., sistemas multinúcleo y de memoria principal). No los abordamos en este capítulo, que se centra en los fundamentos, pero ofrecemos algunas sugerencias en las Notas Bibliográficas.

5.7 Notas bibliográficas

La gestión de transacciones ha sido objeto de considerable estudio desde que los SGBD se han convertido en un área de investigación importante. Existen dos excelentes libros sobre el tema: [Gray y Reuter, 1993] y [Weikum y Vossen, 2001]. Textos clásicos que se centran en estos temas son [Hadzilacos, 1988] y [Bernstein et al., 1987]. Un excelente complemento a estos es [Bernstein y Newcomer, 1997], que ofrece un análisis exhaustivo de los principios del procesamiento de transacciones. También ofrece una visión del procesamiento de transacciones y de los monitores de transacciones más general que la visión centrada en bases de datos que ofrecemos en este libro. Una obra muy importante es el conjunto de notas sobre sistemas operativos de bases de datos de Gray [1979]. Estas notas contienen información valiosa sobre la gestión de transacciones, entre otros temas.

El control de concurrencia distribuida se aborda extensamente en [Bernstein y Goodman, 1981], obra actualmente agotada, pero accesible en línea. Los temas abordados en este capítulo se discuten con mucho más detalle en [Cellary et al. 1988, Bernstein y otros, 1987, Papadimitriou, 1986] y [Gray y Reuter, 1993].

Para las técnicas fundamentales que hemos analizado en este artículo, la 2PL centralizada fue propuesta inicialmente por Alsberg y Day [1976], la detección de interbloqueos jerárquicos fue descrita por Menasce y Muntz [1979], mientras que la detección de interbloqueos distribuidos se debe a Obermack [1982]. Nuestro análisis del algoritmo TO conservador se debe a Herman y Verjus [1979]. El algoritmo TO multiversión original fue propuesto por Reed [1978], con una formalización posterior por Bernstein y Goodman [1983]. Lomet et al. [2012] analizan cómo implementar el multiversionado sobre una capa de concurrencia que implementa 2PL, mientras que Faleiro y Abadi [2015] hacen lo mismo sobre una que implementa TO. También existen enfoques que implementan el versionado como

marco sobre cualquier técnica de control de concurrencia [Agrawal y Sengupta, 1993]. Bernstein et al. [1987] analizan cómo implementar algoritmos de control de concurrencia optimista basados en bloqueos, mientras que Thomas [1979] y Kung y Robinson [1981] analizan implementaciones basadas en marcas de tiempo. Nuestra discusión en la sección 5.2.4 se debe a Ceri y Owicki [1982]. La propuesta original de aislamiento de instantáneas es de Berenson et al. [1995]. Nuestra discusión del algoritmo de aislamiento de instantáneas se debe a Chairunnanda et al. [2014]. Binnig et al. [2014] analizan la optimización que destacamos al final de esa sección. Los protocolos de aborto presunto y confirmación presuntuosa fueron propuestos por Mohan y Lindsay [1983] y Mohan et al. [1986].

Las fallas del sitio y su capacidad de recuperación son el tema de [Skeen y Stonebraker 1983] y [Skeen 1981]. Este último también propone el algoritmo 3PC y su análisis. Los protocolos de selección de coordinadores que analizamos se deben a Hammer y Shipman [1980] y García-Molina [1982]. Un estudio preliminar de la consistencia en presencia de particionamiento de red es de Davidson et al. [1985]. Thomas [1979] propuso la técnica original de votación por mayoría, y la versión no replicada del protocolo que analizamos se debe a Skeen [1982a]. La idea del registro de transacciones distribuidas, en la sección 5.4.6, se debe a Bernstein et al. [1987] y Lampson y Sturgis [1976].

La discusión sobre la gestión de transacciones en el Sistema R* se debe a Mohan et al. [1986]; NonStop SQL se presenta en [Tandem 1987, 1988, Borr 1988]. Bernstein et al. [1980b] analizan SDD-1 en detalle. Los sistemas más modernos Spanner y LeanXcale, analizados en la sección 5.5, se describen en [Corbett et al. 2013] y [Jimenez-Peris y Patiño Martinez 2011], respectivamente.

Se analizan modelos de transacciones avanzados y se ofrecen diversos ejemplos en [Elmagarmid, 1992]. Las transacciones anidadas también se abordan en [Lynch et al., 1993]. Las transacciones anidadas cerradas se deben a Moss [1985], mientras que los modelos de transacciones anidadas abiertas son propuestos por García-Molina y Salem [1987], García-Molina et al. [1990] y las transacciones divididas por Pu [1988]. Los modelos de transacciones anidadas y sus algoritmos específicos de control de concurrencia han sido objeto de algunos estudios. Se pueden encontrar resultados específicos en [Moss 1985, Lynch 1983b, Lynch y Merritt 1986, Fekete et al. 1987a,b, Goldman 1987, Beeri et al. 1989, Fekete et al. 1989] y en [Lynch et al. 1993]. Georgakopoulos et al. [1995] ofrecen una buena introducción a los sistemas de flujo de trabajo, y el tema se aborda en [Dogac et al. 1998] y [van Hee 2002].

El trabajo sobre gestión de transacciones con conocimiento semántico se presenta en [Lynch 1983a, García-Molina 1983] y [Farrag y Özsü 1989]. El procesamiento de transacciones de solo lectura se analiza en [García-Molina y Wiederhold 1982]. Los grupos de transacciones [Skarra et al. 1986, Skarra 1989] también explotan un criterio de corrección denominado patrones semánticos, que es más flexible que la serialización. Además, el trabajo sobre el sistema ARIES [Haderle et al. 1992] también se enmarca en esta clase de algoritmos. En particular, [Rothermel y Mohan 1989] analizan ARIES en el contexto de transacciones anidadas. La serialización de Epsilon [Ramamritham y Pu 1995, Wu et al. 1997] y el modelo NT/PV [Kshemkalyani y Singhal 1994] son otros criterios de corrección menos rigurosos. En [Halici y Dogac 1989] se describe un algoritmo basado en la ordenación de transacciones mediante números de serialización .

Dos libros se centran en el rendimiento de los mecanismos de control de concurrencia, con especial atención a los sistemas centralizados [Kumar 1996, Thomasian 1996]. Kumar [1996] se centra en el rendimiento de los SGBD centralizados; el rendimiento de los métodos de control de concurrencia distribuidos se analiza en [Thomasian 1996] y [Cellary et al.].

1988]. Una revisión temprana pero exhaustiva de la gestión de interbloqueos se encuentra en [Islor y Marsland 1980]. La mayor parte del trabajo sobre la gestión distribuida de interbloqueos se ha centrado en la detección y resolución (véase, por ejemplo, [Obermack 1982, Elmagarmid et al. 1988]). Se incluyen estudios de los algoritmos importantes en [Elmagarmid 1986], [Knapp 1987] y [Singhal 1989].

El aislamiento de instantáneas ha recibido mucha atención en los últimos años. Aunque Oracle había implementado SI desde sus primeras versiones, el concepto se definió formalmente en [Berenson et al. 1995]. Una línea de trabajo que no cubrimos en este capítulo es cómo obtener una ejecución serializable incluso cuando se utiliza SI como criterio de corrección. Esta línea de trabajo modifica el algoritmo de control de concurrencia detectando las anomalías causadas por SI que conducen a la inconsistencia de los datos y previniéndolas [Cahill et al. 2009, Alomari et al. 2009, Revilak et al. 2011, Alomari et al. 2008] y estas técnicas han comenzado a incorporarse en sistemas, por ejemplo, PostgreSQL [Ports y Grittner 2012]. El primer algoritmo de control de concurrencia de SI se debe a Schenkel et al. [2000], centrándose en el control de concurrencia en sistemas de integración de datos que utilizan SI. En este capítulo basamos nuestra discusión en el sistema ConfluxDB [Chairunnanda et al. 2014]; otro trabajo en esta dirección es el de Binnig et al. [2014], donde se desarrollan técnicas más refinadas.

Kohler [1981] presenta un análisis general de los problemas de confiabilidad en sistemas de bases de datos distribuidas. Hadzilacos [1988] formaliza el concepto de confiabilidad.

Los aspectos de confiabilidad del Sistema R* se detallan en [Traiger et al. 1982], mientras que Hammer y Shipman [1980] describen lo mismo para el sistema SDD-1.

Se puede encontrar material más detallado sobre las funciones del gestor de recuperación local en [Verhofstadt 1978, Härder y Reuter 1983]. La implementación de las funciones de recuperación local en System R se describe en [Gray et al. 1981].

El protocolo de confirmación de dos fases se describe por primera vez en [Gray, 1979]. Se presentan modificaciones en [Mohan y Lindsay, 1983]. La definición de confirmación de tres fases se debe a Skeen [1981, 1982b]. Los resultados formales sobre la existencia de protocolos de terminación no bloqueantes se deben a Skeen y Stonebraker [1983].

Paxos fue propuesto originalmente por Lamport [1998]. Este documento se considera difícil de leer, lo que ha dado lugar a diversos artículos que describen el protocolo.

Lamport [2001] ofrece una descripción significativamente simplificada, mientras que Van Renesse y Altinbuken [2015] ofrecen una descripción que quizás se encuentra entre estos dos puntos y que constituye un buen artículo para estudiar. El Paxos 2PC que destacamos brevemente es propuesto por Gray y Lamport [2006]. Para una discusión sobre cómo diseñar un sistema basado en Paxos, se recomiendan [Chandra et al. 2007] y [Kirsch y Amir 2008]. Existen muchas versiones diferentes de Paxos —demasiadas para enumerarlas aquí—, lo que ha llevado a que se le denomine una "familia de protocolos". No proporcionamos referencias a cada una de ellas. También observamos que Paxos no es el único algoritmo de consenso; se han propuesto diversas alternativas, especialmente con la popularización de las cadenas de bloques (véase nuestra discusión sobre cadenas de bloques en el capítulo 9). Esta lista crece rápidamente, razón por la cual

No proporcionamos referencias. Se ha propuesto un algoritmo, Raft, en respuesta a la complejidad y la dificultad percibida para comprender Paxos. La propuesta original es de Ongaro y Ousterhout [2014] y se describe detalladamente en el capítulo 23 de [Silberschatz et al., 2019].

Como se mencionó anteriormente, en este capítulo no abordamos los fallos bizantinos. El protocolo de Paxos tampoco los aborda. Castro y Liskov [1999] ofrecen una buena descripción de cómo abordar este tipo de fallos.

En cuanto a trabajos relevantes más recientes, Tu et al. [2013] analizan el procesamiento de transacciones a gran escala en una sola máquina multinúcleo. Kemper y Neumann [2011] analizan problemas de gestión de transacciones en un entorno híbrido OLAP/OLTP dentro del contexto del sistema de memoria principal HyPer. De forma similar, Larson et al. [2011] analizan el mismo problema dentro del contexto del sistema Hekaton. El sistema E-store que analizamos en el capítulo 2, como parte del particionamiento adaptativo de datos, también aborda la gestión de transacciones en DBMS distribuidos y particionados. Como se indica allí, E-store utiliza Squall [Elmore et al. 2015], que considera las transacciones al decidir el movimiento de datos. Thomson y Abadi [2010] proponen Calvin, que combina una técnica de prevención de interbloqueos con algoritmos de control de concurrencia para obtener historiales que garantizan ser equivalentes a un ordenamiento serial predeterminado en DBMS distribuidos y replicados.

Ceremonias

Problema 5.1 ¿Cuáles de las siguientes historias son equivalentes a conflictos?

$$H1 = \{W2(x), W1(x), R3(x), R1(x), W2(y), R3(y), R3(z), R2(x)\}$$

$$H2 = \{R3(z), R3(y), W2(y), R2(z), W1(x), R3(x), W2(x), R1(x)\}$$

$$H3 = \{R3(z), W2(x), W2(y), R1(x), R3(x), R2(z), R3(y), W1(x)\}$$

$$H4 = \{R2(z), W2(x), W2(y), W1(x), R1(x), R3(x), R3(z), R3(y)\}$$

Problema 5.2 ¿Cuáles de las historias anteriores H1 – H4 son serializables?

Problema 5.3 Proporcione un historial de dos transacciones completas que no está permitido por un programador 2PL estricto pero es aceptado por el programador 2PL básico.

Problema 5.4 (*) Se dice que el historial H es recuperable si, siempre que la transacción Ti lee (algún elemento x) de la transacción Tj (i = j) en H y Ci ocurre en H, entonces Cj ⊂ Ci. Ti "lee x de" Tj en H si 1. Wj(x) ⊂ H Ri(x), y 2. Aj no ⊂ H Ri(x), y 3. si

hay algún Wk(x) tal que Wj(x)

H Wk(x) ⊂ H Ri(x), entonces

Ak ⊂ H

Ri(x).

¿Cuáles de las siguientes historias son recuperables?

$$H1 = \{W2(x), W1(x), R3(x), R1(x), C1, W2(y), R3(y), R3(z), C3, R2(x), C2\}$$

$$H2 = \{R3(z), R3(y), W2(y), R2(z), W1(x), R3(x), W2(x), R1(x), C1, C2, C3\}$$

$$H3 = \{R3(z), W2(x), W2(y), R1(x), R3(x), R2(z), R3(y), C3, W1(x), C2, C1\}$$

$$H4 = \{R2(z), W2(x), W2(y), C2, W1(x), R1(x), A1, R3(x), R3(z), R3(y), C3\}$$

Problema 5.5 (*) Proporcione los algoritmos para los administradores de transacciones y los administradores de bloqueo para el enfoque de bloqueo distribuido de dos fases.

Problema 5.6 ()** Modificar el algoritmo 2PL centralizado para manejar la lectura fantasma. La lectura fantasma ocurre cuando se ejecutan dos lecturas dentro de una transacción y el resultado devuelto por la segunda lectura contiene tuplas que no existen en la primera.

Considere el siguiente ejemplo, basado en la base de datos de reservas de aerolíneas que se describió al principio de este capítulo: La transacción T1, durante su ejecución, busca en la tabla FC los nombres de los clientes que han pedido una comida especial. Obtiene un conjunto de CNAME para los clientes que cumplen los criterios de búsqueda. Mientras T1 se ejecuta, la transacción T2 inserta nuevas tuplas en FC con la solicitud de comida especial y realiza la confirmación. Si T1 volviera a emitir la misma consulta de búsqueda posteriormente, obtendría un conjunto de CNAME diferente del conjunto original recuperado. Por lo tanto, han aparecido tuplas "fantasma" en la base de datos.

Problema 5.7. Los algoritmos de control de concurrencia basados en la ordenación de marcas de tiempo dependen de un reloj preciso en cada sitio o de un reloj global al que todos los sitios pueden acceder (el reloj puede ser un contador). Supongamos que cada sitio tiene su propio reloj que marca cada 0,1 segundos. Si todos los relojes locales se resincronizan cada 24 horas, ¿cuál es la desviación máxima en segundos por 24 horas permitida en cualquier sitio local para garantizar que un mecanismo basado en marcas de tiempo sincronice correctamente las transacciones?

Problema 5.8 ()** Incorpore la estrategia de bloqueo distribuido descrita en este capítulo en los algoritmos 2PL distribuidos que diseñó en el problema [5.5](#).

Problema 5.9 Explique la relación entre el requisito de almacenamiento del administrador de transacciones y el tamaño de la transacción (número de operaciones por transacción) para un administrador de transacciones que utiliza un ordenamiento de marca de tiempo optimista para el control de concurrencia.

Problema 5.10 (*) Proporcione los algoritmos del programador y del administrador de transacciones para el controlador de concurrencia optimista distribuida descrito en este capítulo.

Problema 5.11. Recuerde, de la discusión en la Sección [5.6](#), que el modelo computacional utilizado en las descripciones de este capítulo es centralizado. ¿Cómo cambiarían los algoritmos del gestor de transacciones 2PL distribuido y del gestor de bloqueos si se utilizara un modelo de ejecución distribuido?

Problema 5.12 A veces se afirma que la serializabilidad es un criterio de corrección bastante restrictivo.

¿Podría dar ejemplos de historiales distribuidos que sean correctos (es decir, que mantengan la consistencia de las bases de datos locales, así como su consistencia mutua) pero que no sean serializables?

Problema 5.13 (*) Analice el protocolo de terminación de falla del sitio para 2PC utilizando una topología de comunicación distribuida.

Problema 5.14 (*)

Diseñe un protocolo 3PC utilizando la topología de comunicación lineal.

Problema 5.15 (*) En nuestra presentación del protocolo de terminación centralizado 3PC, el primer paso consiste en enviar el estado del coordinador a todos los participantes. Los participantes cambian de estado según el estado del coordinador. Es posible diseñar el protocolo de terminación de forma que el coordinador, en lugar de enviar su propia información de estado a los participantes, les pida que la envíen al coordinador. Modifique el protocolo de terminación para que funcione de esta manera.

manera.

Problema 5.16 (**) En la sección 5.4.6 afirmamos que un planificador que implementa un algoritmo de control de concurrencia estricto siempre estará listo para confirmar una transacción al recibir el mensaje de "preparación" del coordinador. Demuestre esta afirmación.

Problema 5.17 (**) Suponiendo que el coordinador se implementa como parte del administrador de transacciones y el participante como parte del programador, proporcione los algoritmos del administrador de transacciones, del programador y del administrador de recuperación local para un DBMS distribuido no replicado bajo los siguientes supuestos.

(a) El programador implementa un algoritmo de control de concurrencia de bloqueo distribuido (estricto) de dos fases. (b) Los

registros del protocolo de confirmación se escriben en un registro de base de datos central por el programador. LRM cuando lo llama el programador.

(c) El LRM puede implementar cualquiera de los protocolos descritos (p. ej., fix/no-flush u otros). Sin embargo, se modifica para admitir los procedimientos de recuperación distribuida, como se explicó en la sección 5.4.6.

Problema 5.18 (*) Escriba los algoritmos detallados para el administrador de recuperación local sin reparación ni descarga.

Problema 5.19 (**) Suponga que

(a) El programador implementa un control de concurrencia de bloqueo de dos fases centralizado, (b) El LRM implementa un protocolo de no reparación/no limpieza.

Proporcione algoritmos detallados para el administrador de transacciones, el programador y los administradores de recuperación local.

Capítulo 6

Replicación de datos



Como comentamos en capítulos anteriores, las bases de datos distribuidas normalmente se replican. Los propósitos de la replicación son múltiples:

1. Disponibilidad del sistema. Como se explicó en el capítulo 1, los SGBD distribuidos pueden eliminar puntos únicos de fallo mediante la replicación de datos, de modo que los datos sean accesibles desde múltiples sitios. En consecuencia, incluso cuando algunos sitios fallan, los datos pueden ser accesibles desde otros.
2. Rendimiento. Como vimos anteriormente, uno de los principales factores que contribuyen al tiempo de respuesta es la sobrecarga de comunicación. La replicación nos permite ubicar los datos más cerca de sus puntos de acceso, localizando así la mayor parte del acceso, lo que contribuye a una reducción del tiempo de respuesta.
3. Escalabilidad. A medida que los sistemas crecen geográficamente y en cuanto al número de sitios (y, por consiguiente, en cuanto al número de solicitudes de acceso), la replicación permite soportar este crecimiento con tiempos de respuesta aceptables.
4. Requisitos de la aplicación. Finalmente, la replicación puede depender de las aplicaciones, que podrían desear mantener múltiples copias de datos como parte de sus especificaciones operativas.

Si bien la replicación de datos ofrece claras ventajas, plantea el considerable reto de mantener sincronizadas las diferentes copias. Abordaremos esto en breve, pero primero consideraremos el modelo de ejecución en bases de datos replicadas. Cada elemento de datos replicado x tiene un número de copias x_1, x_2, \dots, x_n . Nos referiremos a x como el elemento de datos lógico y a sus copias (o réplicas) como elementos de datos físicos.¹ Para garantizar la transparencia de la replicación, las transacciones de usuario ejecutarán operaciones de lectura y escritura en el elemento de datos lógico x . El protocolo de control de réplicas se encarga de asignar estas operaciones a lecturas y escrituras en los elementos de datos físicos x_1, \dots, x_n . Por lo tanto, el sistema se comporta como si existiera una única copia de cada elemento de datos, lo que se conoce como imagen única del sistema o equivalencia de una copia. La implementación específica de las interfaces de lectura y escritura

¹En este capítulo, utilizamos los términos “réplica”, “copia” y “elemento de datos físicos” indistintamente.

El funcionamiento del monitor de transacciones varía según el protocolo de replicación específico, y analizaremos estas diferencias en las secciones correspondientes.

Existen diversas decisiones y factores que influyen en el diseño de los protocolos de replicación. Algunos de ellos se analizaron en capítulos anteriores, mientras que otros se abordarán aquí.

Diseño de bases de datos. Como se explicó en el capítulo 2, una base de datos distribuida puede estar total o parcialmente replicada. En el caso de una base de datos parcialmente replicada, el número de elementos de datos físicos para cada elemento de datos lógicos puede variar, e incluso algunos elementos de datos pueden no estar replicados. En este caso, las transacciones que acceden únicamente a elementos de datos no replicados son transacciones locales (ya que pueden ejecutarse localmente en un sitio) y su ejecución no suele ser relevante en este contexto. Las transacciones que acceden a elementos de datos replicados deben ejecutarse en múltiples sitios y son transacciones globales.

- Consistencia de la base de datos. Cuando las transacciones globales actualizan copias de un elemento de datos en diferentes sitios, los valores de estas copias pueden variar en un momento dado. Se dice que una base de datos replicada es mutuamente consistente si todas las réplicas de cada uno de sus elementos de datos tienen valores idénticos. Lo que diferencia los distintos criterios de consistencia mutua reside en la precisión con la que deben sincronizarse las réplicas. Algunos garantizan la coherencia mutua de las réplicas al confirmar una transacción de actualización; por lo tanto, se suelen denominar criterios de consistencia fuerte . Otros adoptan un enfoque más flexible y se denominan criterios de consistencia débil .

Dónde se realizan las actualizaciones. Una decisión fundamental al diseñar un protocolo de replicación es dónde se realizan primero las actualizaciones de la base de datos. Las técnicas se pueden caracterizar como centralizadas si realizan las actualizaciones primero en una copia maestra , a diferencia de las distribuidas si permiten actualizaciones en cualquier réplica.

Las técnicas centralizadas pueden identificarse además como una copia maestra única , cuando solo hay una copia de la base de datos maestra en el sistema, o una copia primaria , donde la copia maestra de cada elemento de datos puede ser diferente.2

Propagación de actualizaciones. Una vez que se realizan actualizaciones en una réplica (maestra o no), la siguiente decisión es cómo se propagan las actualizaciones a las demás. Las alternativas se identifican como técnicas diligentes o perezosas. Las técnicas diligentes realizan todas las actualizaciones dentro del contexto de la transacción global que inició las operaciones de escritura. Por lo tanto, cuando la transacción se confirma, sus actualizaciones se habrán aplicado a todas las copias. Las técnicas perezosas, por otro lado, propagan las actualizaciones poco después de que se haya confirmado la transacción inicial.

Las técnicas ansiosas se identifican además según cuándo envían cada escritura a las otras réplicas: algunas envían cada operación de escritura individualmente, otras agrupan las escrituras y las propagan en el punto de confirmación.

En la literatura, las técnicas centralizadas se denominan «maestro único», mientras que las distribuidas se denominan «multimáestro» o «actualización en cualquier lugar». Estos términos, en particular «maestro único», resultan confusos, ya que se refieren a arquitecturas alternativas para implementar protocolos centralizados (más información en la sección 6.2.3). Por lo tanto, preferimos los términos más descriptivos «centralizado» y «distribuido».

- Grado de transparencia de replicación. Algunos protocolos de replicación requieren que cada aplicación de usuario conozca el sitio maestro donde se enviarán las transacciones. Estos protocolos solo ofrecen una transparencia de replicación limitada a las aplicaciones de usuario. Otros protocolos ofrecen una transparencia de replicación total al involucrar a la TM en cada sitio. En este caso, las aplicaciones de usuario envían las transacciones a sus TM locales en lugar de al sitio maestro.

En la sección 6.1, abordamos los problemas de consistencia en bases de datos replicadas y, en la sección 6.2, analizamos la aplicación de actualizaciones centralizada frente a la distribuida, así como las alternativas de propagación de actualizaciones. Esto nos llevará a la sección 6.3, donde abordaremos los protocolos específicos. En la sección 6.4, analizamos el uso de primitivas de comunicación grupal para reducir la sobrecarga de mensajería de los protocolos de replicación. En estas secciones, asumiremos que no se producen fallos para poder centrarnos en los protocolos de replicación. Luego presentaremos las fallas e investigaremos cómo se revisan los protocolos para manejarlas en la Sección 6.5.

6.1 Consistencia de las bases de datos replicadas

Existen dos problemas relacionados con la consistencia de una base de datos replicada. Uno es la consistencia mutua, como se mencionó anteriormente, que aborda la convergencia de los valores de los elementos de datos físicos correspondientes a un elemento de datos lógico. El segundo es la consistencia transaccional, como se explicó en el capítulo 5. La serializabilidad, que se introdujo como criterio de consistencia transaccional, debe reformularse en el caso de las bases de datos replicadas. Además, existen relaciones entre la consistencia mutua y la consistencia transaccional. En esta sección, primero analizamos los enfoques de consistencia mutua y luego nos centramos en la redefinición de la consistencia transaccional y su relación con la consistencia mutua.

6.1.1 Coherencia mutua

Como se indicó anteriormente, los criterios de consistencia mutua para bases de datos replicadas pueden ser fuertes o débiles. Cada criterio es adecuado para diferentes tipos de aplicaciones con distintos requisitos de consistencia.

Los criterios de consistencia mutua sólida exigen que todas las copias de un elemento de datos tengan el mismo valor al finalizar la ejecución de una transacción de actualización. Esto se logra mediante diversos métodos, pero la ejecución de 2PC en el punto de confirmación de una transacción de actualización es una forma común de lograr una consistencia mutua sólida.

Los criterios de consistencia mutua débil no exigen que los valores de las réplicas de un elemento de datos sean idénticos al finalizar una transacción de actualización. Lo que se requiere es que, si la actividad de actualización cesa durante un tiempo, los valores eventualmente se vuelvan idénticos.

Esto se conoce comúnmente como consistencia eventual, que se refiere al hecho de que

Los valores de las réplicas pueden divergir con el tiempo, pero eventualmente convergerán. Es difícil definir este concepto formalmente o con precisión, aunque la siguiente definición de Saito y Shapiro es probablemente la más precisa posible:

Un [elemento de datos] replicado es eventualmente consistente cuando cumple las siguientes condiciones, asumiendo que todas las réplicas comienzan desde el mismo estado inicial.

- En cualquier momento, para cada réplica, existe un prefijo del [historial] equivalente al prefijo del [historial] de todas las demás réplicas. A esto lo llamamos prefijo comprometido de la réplica.

El prefijo comprometido de cada réplica crece monótonamente con el tiempo. Todas las operaciones no abortadas en el prefijo comprometido cumplen sus precondiciones. Por cada operación α enviada, α o [su abortado] se incluirán eventualmente en el prefijo comprometido.

Cabe señalar que esta definición de consistencia eventual es bastante estricta, en particular los requisitos de que los prefijos históricos sean los mismos en todo momento y que el prefijo comprometido crezca monótonamente. Muchos sistemas que afirman proporcionar consistencia eventual incumplirían estos requisitos.

La serialización de épsilon (ESR) permite que una consulta vea datos inconsistentes mientras se actualizan las réplicas, pero requiere que las réplicas converjan a un estado equivalente a una copia una vez que las actualizaciones se propagan a todas las copias. Limita el error en los valores de lectura mediante un valor de épsilon (), que se define en términos del número de actualizaciones (operaciones de escritura) que una consulta "pierde". Dada una transacción (consulta) de solo lectura TQ, sea TU el conjunto de todas las transacciones de actualización que se ejecutan simultáneamente con TQ. Si $RS(TQ) \cap WS(TU) = \emptyset$ (TQ está leyendo alguna copia de algunos elementos de datos mientras una transacción en TU está actualizando (posiblemente una copia diferente) de esos elementos de datos), entonces hay un conflicto de lectura-escritura y TQ puede estar leyendo datos inconsistentes. La inconsistencia está limitada por los cambios realizados por TU . Claramente, la ESR no sacrifica la consistencia de la base de datos, sino que solo permite que las transacciones de solo lectura (consultas) lean datos inconsistentes. Por esta razón, se ha afirmado que la ESR no debilita la consistencia de la base de datos, sino que la "estira".

También se han debatido otros límites más flexibles. Incluso se ha sugerido que los usuarios puedan especificar restricciones de frescura adecuadas para aplicaciones específicas, y que los protocolos de replicación las apliquen. Los tipos de restricciones de frescura que se pueden especificar son los siguientes:

- Restricciones temporales. Los usuarios pueden aceptar la divergencia de los valores de las copias físicas hasta un cierto intervalo de tiempo: x_i puede reflejar el valor de una actualización en el tiempo t , mientras que x_j puede reflejar el valor en $t - \Delta t$, lo cual puede ser aceptable. • Restricciones de valor. Puede ser aceptable que los valores de todos los elementos de datos físicos se encuentren dentro de un cierto rango entre sí. El usuario puede considerar que la base de datos es mutuamente consistente si los valores no divergen más de una cierta cantidad (o porcentaje). •

Restricciones de desviación en múltiples elementos de datos. En las transacciones que leen múltiples elementos de datos, los usuarios pueden estar satisfechos si la desviación temporal entre las marcas de tiempo de actualización de dos elementos de datos es menor que un umbral (es decir, se actualizaron dentro de ese umbral) o, en el caso del cálculo agregado, si el cálculo agregado

sobre un elemento de datos está dentro de un cierto rango del valor más reciente (es decir, incluso si los valores de las copias físicas individuales pueden estar más desincronizados que este rango, siempre que un cálculo agregado particular esté dentro del rango, puede ser aceptable).

Un criterio importante al analizar protocolos que emplean criterios que permiten la divergencia de las réplicas es el grado de actualización. El grado de actualización de una réplica dada x_i en el tiempo t se define como la proporción de actualizaciones aplicadas en x_i en el tiempo t respecto al número total de actualizaciones.

6.1.2 Coherencia mutua versus consistencia de transacción

La consistencia mutua, tal como la definimos aquí, y la consistencia transaccional, como se explicó en el capítulo 5, están relacionadas, pero son diferentes. La consistencia mutua se refiere a que las réplicas convergen al mismo valor, mientras que la consistencia transaccional requiere que el historial de ejecución global sea serializable. Un SGBD replicado puede garantizar que los elementos de datos sean mutuamente consistentes al confirmar una transacción, pero el historial de ejecución podría no ser serializable globalmente. Esto se demuestra en el siguiente ejemplo.

Ejemplo 6.1 Considerese tres sitios (A, B y C) y tres elementos de datos (x, y, z) distribuidos de la siguiente manera: el sitio A aloja x , el sitio B aloja x, y , el sitio C aloja x, y, z . Utilizaremos identificadores de sitio como subíndices en los elementos de datos para hacer referencia a una réplica específica.

Consideremos ahora las siguientes tres transacciones:

T1: $x \leftarrow 20$	T2: Leer(x)	T3: Leer(x)
Escribe(x) $y \leftarrow x + y$		Leer(y) z
Comprometerse	Escribe(y)	$\leftarrow (x + y)/100$
	Comprometerse	Escribe(z)
		Comprometerse

Tenga en cuenta que la escritura de T1 debe ejecutarse en los tres sitios (ya que x se replica en los tres sitios), la escritura de T2 debe ejecutarse en B y C, y la escritura de T3 debe ejecutarse solo en C. Estamos asumiendo un modelo de ejecución de transacciones donde las transacciones pueden leer sus réplicas locales, pero tienen que actualizar todas las réplicas.

Supongamos que las siguientes tres historias locales se generan en los sitios:

$$HA = \{W1(x_A), C1\}$$

$$HB = \{W1(x_B), C1, R2(x_B), W2(y_B), C2\}$$

$$HC = \{W2(y_C), C2, R3(x_C), R3(y_C), W3(z_C), C3, W1(x_C), C1\}$$

El orden de serialización en HB es $T1 \rightarrow T2$, mientras que en HC es $T2 \rightarrow T3 \rightarrow T1$.

Por lo tanto, el historial global no es serializable. Sin embargo, la base de datos es mutuamente consistente. Supongamos, por ejemplo, que inicialmente $x_A = x_B = x_C = 10, y_B = y_C =$

15, y $zC = 7$. Con los historiales anteriores, los valores finales serán $xA = xB = xC = 20$, $yB = yC = 35$, $zC = 3,5$. Todas las copias físicas (réplicas) tienen convergieron al mismo valor.

Por supuesto, es posible que las bases de datos sean mutuamente inconsistentes y El historial de ejecución no se puede serializar globalmente, como se muestra en lo siguiente ejemplo.

Ejemplo 6.2 Considere dos sitios (A y B) y un elemento de datos (x) que se replica En ambos sitios (xA y xB). Consideremos además las dos transacciones siguientes:

T1: Leer(x)	$x \leftarrow x$	T2: Leer(x)
+ 5		$x \leftarrow x \quad 10$
Escribe(x)		Escribe(x)
Comprometerse		Comprometerse

Supongamos que las dos historias locales siguientes se generan en los dos sitios (de nuevo utilizando el modelo de ejecución del ejemplo anterior):

$$HA = \{R1(xA), W1(xA), C1, R2(xA), W2(xA), C2\}$$

$$HB = \{R2(xB), W2(xB), C2, R1(xB), W1(xB), C1\}$$

Aunque ambas historias son seriales, serializan T1 y T2 en sentido inverso. orden; por lo tanto, el historial global no es serializable. Además, también se viola la consistencia mutua. Suponga que el valor de x antes de la ejecución de estos transacciones fue 1. Al final de la ejecución de estos cronogramas, el valor de x es 60 en el sitio A, mientras que es 15 en el sitio B. Por lo tanto, en este ejemplo, el historial global es no serializables y las bases de datos son mutuamente inconsistentes.

Dada la observación anterior, el criterio de consistencia de la transacción dado en El capítulo 5 se amplía en bases de datos replicadas para definir la serialización de una copia. La serialización de una copia (1SR) establece que los efectos de las transacciones en los datos replicados Los elementos deben ser los mismos que si se hubieran realizado uno a la vez en un solo conjunto. de elementos de datos. En otras palabras, los historiales son equivalentes a una ejecución en serie. sobre elementos de datos no replicados.

El aislamiento de instantáneas que presentamos en el capítulo 5 también se ha extendido para bases de datos replicadas y se ha utilizado como un criterio de consistencia transaccional alternativo dentro el contexto de bases de datos replicadas. De manera similar, una forma más débil de serialización, llamada Se ha definido la serialización de concurrencia relajada (RC) que corresponde al nivel de aislamiento de lectura confirmada.

6.2 Estrategias de gestión de actualizaciones

Como se mencionó anteriormente, los protocolos de replicación se pueden clasificar según cuándo Las actualizaciones se propagan a copias (ansiosas versus perezosas) y donde se realizan las actualizaciones.

Permitido que ocurra (centralizado versus distribuido). Estas dos decisiones se conocen generalmente como estrategias de gestión de actualizaciones . En esta sección, analizamos estas alternativas antes de presentar los protocolos en la siguiente.

6.2.1 Propagación de actualizaciones ansiosas

Los enfoques de propagación de actualizaciones diligentes aplican los cambios a todas las réplicas dentro del contexto de la transacción de actualización. Por lo tanto, cuando la transacción de actualización se confirma, todas las copias tienen el mismo valor. Normalmente, las técnicas de propagación diligente utilizan 2PC en el punto de confirmación, pero, como veremos más adelante, existen alternativas para lograr un acuerdo. Además, la propagación diligente puede utilizar la propagación síncrona de cada actualización, aplicándola a todas las réplicas simultáneamente (al emitirse la escritura), o la propagación diferida , donde las actualizaciones se aplican a una réplica al emitirse, pero su aplicación a las demás se agrupa y se difiere hasta el final de la transacción. La propagación diferida puede implementarse incluyendo las actualizaciones en el mensaje "Preparación para confirmar" al inicio de la ejecución de 2PC.

Las técnicas ansiosas suelen aplicar criterios estrictos de consistencia mutua. Dado que todas las réplicas son mutuamente consistentes al final de una transacción de actualización, una lectura posterior puede leer desde cualquier copia (es decir, se puede asignar un $R(x)$ a $R(x_i)$ para cualquier x_i). Sin embargo, se debe aplicar un $W(x)$ a todas las réplicas (es decir, $W(x_1), \dots, x_n$). Por lo tanto, los protocolos que siguen la propagación de actualizaciones ansiosas se conocen como protocolos de lectura única/escritura total (ROWA).

La propagación de actualizaciones diligentes tiene tres ventajas. En primer lugar, suele garantizar la consistencia mutua mediante 1SR; por lo tanto, no hay inconsistencias transaccionales. En segundo lugar, una transacción puede leer una copia local del dato (si existe una copia local) y garantizar que se lea un valor actualizado.

Por lo tanto, no es necesario realizar una lectura remota. Finalmente, los cambios en las réplicas se realizan de forma automática; por lo tanto, la recuperación ante fallos puede controlarse mediante los protocolos que ya estudiamos en el capítulo anterior.

La principal desventaja de la propagación de actualizaciones diligentes es que una transacción debe actualizar todas las copias antes de poder finalizar. Esto tiene dos consecuencias. En primer lugar, el rendimiento del tiempo de respuesta de la transacción de actualización se ve afectado, ya que normalmente debe participar en una ejecución de 2PC y porque la velocidad de actualización está limitada por la máquina más lenta. En segundo lugar, si una de las copias no está disponible, la transacción no puede finalizar, ya que todas las copias deben actualizarse. Como se explicó en el capítulo 5, si es posible diferenciar entre fallos de sitio y fallos de red, se puede finalizar la transacción siempre que solo una réplica no esté disponible (recuerde que la indisponibilidad de más de un sitio provoca el bloqueo de 2PC), pero generalmente no es posible diferenciar entre estos dos tipos de fallos.

6.2.2 Propagación de actualizaciones diferidas

En la propagación de actualizaciones diferida, no todas las actualizaciones de las réplicas se realizan dentro del contexto de la transacción de actualización. En otras palabras, la transacción no espera a que sus actualizaciones se apliquen a todas las copias antes de confirmarse, sino que se confirma en cuanto se actualiza una réplica. La propagación a otras copias se realiza de forma asíncrona desde la transacción original, mediante transacciones de actualización que se envían a los sitios de réplica un tiempo después de confirmarse la transacción de actualización. Una transacción de actualización contiene la secuencia de actualizaciones de la transacción de actualización correspondiente.

La propagación perezosa se utiliza en aplicaciones donde la consistencia mutua fuerte puede ser innecesaria y demasiado restrictiva. Estas aplicaciones pueden tolerar cierta inconsistencia entre las réplicas a cambio de un mejor rendimiento.

Ejemplos de estas aplicaciones son el Servicio de Nombres de Dominio (DNS), bases de datos de sitios geográficamente distribuidos, bases de datos móviles y bases de datos de asistentes digitales personales. En estos casos, suele aplicarse una consistencia mutua débil.

La principal ventaja de las técnicas de propagación de actualizaciones diferidas es que generalmente ofrecen tiempos de respuesta más bajos para las transacciones de actualización, ya que una transacción de actualización puede confirmarse en cuanto actualiza una copia. Las desventajas son que las réplicas no son consistentes entre sí y algunas pueden estar desactualizadas. Por consiguiente, una lectura local puede leer datos obsoletos y no garantiza la devolución del valor actualizado. Además, en algunos escenarios que analizaremos más adelante, es posible que las transacciones no vean sus propias escrituras; es decir, $R_i(x)$ de una transacción de actualización T_i podría no ver los efectos de $W_j(x)$ ejecutada previamente. Esto se conoce como inversión de transacción. La serialización de una copia fuerte (1SR fuerte) y el aislamiento de instantáneas fuerte (SI fuerte) impiden todas las inversiones de transacción en los niveles de aislamiento 1SR y SI, respectivamente, pero son costosos de proporcionar. Las garantías más débiles de 1SR y SI global, si bien son mucho más económicas que sus contrapartes más fuertes, no impiden las inversiones de transacción. Se han propuesto garantías transaccionales a nivel de sesión en los niveles de aislamiento 1SR y SI que abordan estas deficiencias al evitar inversiones de transacciones dentro de una sesión de cliente, pero no necesariamente entre sesiones. Estas garantías a nivel de sesión son menos costosas de proporcionar que sus contrapartes robustas, a la vez que conservan muchas de sus propiedades deseables.

6.2.3 Técnicas centralizadas

Las técnicas de propagación centralizada de actualizaciones requieren que las actualizaciones se apliquen primero en una copia maestra y luego se propaguen a otras copias (denominadas copias esclavas). El sitio que aloja la copia maestra se denomina también sitio maestro, mientras que los sitios que alojan las copias esclavas de ese elemento de datos se denominan sitios esclavos.

En algunas técnicas, existe un único maestro para todos los datos replicados. Estas se denominan técnicas centralizadas de un solo maestro. En otros protocolos, la copia maestra...

Para cada dato, la configuración puede ser diferente (es decir, para el dato x , la copia maestra puede ser x_i , almacenada en el sitio S_i , mientras que para el dato y , puede ser y_j , almacenada en el sitio S_j). Estas técnicas se conocen comúnmente como técnicas centralizadas de copia primaria .

Las técnicas centralizadas tienen dos ventajas. En primer lugar, la aplicación de las actualizaciones es sencilla, ya que se realizan únicamente en el sitio maestro y no requieren sincronización entre varios sitios de réplica. En segundo lugar, se garantiza que al menos un sitio (el que contiene la copia maestra) tenga valores actualizados para un elemento de datos. Estos protocolos suelen ser adecuados en almacenes de datos y otras aplicaciones donde el procesamiento de datos se centraliza en uno o varios sitios maestros.

La principal desventaja es que, como en cualquier algoritmo centralizado, si existe un sitio central que aloja todos los maestros, este puede sobrecargarse y convertirse en un cuello de botella. Distribuir la responsabilidad del sitio maestro por cada elemento de datos, como en las técnicas de copia primaria, es una forma de reducir esta sobrecarga, pero plantea problemas de consistencia, en particular con respecto a mantener la serialización global en técnicas de replicación diferida, ya que las transacciones de actualización deben ejecutarse en las réplicas en el mismo orden de serialización. Estos temas se abordan con más detalle en las secciones pertinentes.

6.2.4 Técnicas distribuidas

Las técnicas distribuidas aplican la actualización a la copia local en el sitio donde se origina la transacción de actualización y, posteriormente, las actualizaciones se propagan a los demás sitios de réplica. Se denominan técnicas distribuidas, ya que diferentes transacciones pueden actualizar distintas copias del mismo dato, ubicadas en diferentes sitios. Son adecuadas para aplicaciones colaborativas con centros de decisión/operación distributivos. Permiten distribuir la carga de forma más uniforme y, si se combinan con técnicas de propagación diferida, pueden proporcionar la máxima disponibilidad del sistema.

Una complicación grave que surge en estos sistemas es que diferentes réplicas de un elemento de datos pueden actualizarse simultáneamente en diferentes sitios (replicantes). Si las técnicas distribuidas se combinan con métodos de propagación ávida, los métodos de control de concurrencia distribuidos pueden abordar adecuadamente el problema de las actualizaciones concurrentes.

Sin embargo, si se utilizan métodos de propagación diferida, las transacciones pueden ejecutarse en diferentes órdenes en distintos sitios, lo que provoca un historial global no 1SR. Además, varias réplicas perderán la sincronización. Para solucionar estos problemas, se aplica un método de conciliación que consiste en deshacer y rehacer transacciones, de modo que la ejecución de las mismas sea la misma en cada sitio. Esto no es sencillo, ya que la conciliación generalmente depende de la aplicación.

6.3 Protocolos de replicación

En la sección anterior, analizamos dos dimensiones según las cuales se pueden clasificar las técnicas de gestión de actualizaciones. Estas dimensiones son ortogonales; por lo tanto, cuatro

Existen combinaciones posibles: centralizada diligente, distribuida diligente, centralizada perezosa y distribuida perezosa. En esta sección, analizamos cada una de estas alternativas. Para simplificar la explicación, asumimos una base de datos completamente replicada, lo que significa que todas las transacciones de actualización son globales. Además, asumimos que cada sitio implementa una técnica de control de concurrencia basada en 2PL.

6.3.1 Protocolos centralizados ansiosos

En el control centralizado de réplicas, un sitio maestro controla las operaciones sobre un elemento de datos. Estos protocolos se combinan con técnicas de consistencia rigurosa, de modo que las actualizaciones de un elemento de datos lógico se aplican a todas sus réplicas dentro del contexto de la transacción de actualización, la cual se confirma mediante el protocolo 2PC (aunque existen alternativas que no son 2PC, como se explicará más adelante). En consecuencia, una vez completada la transacción de actualización, todas las réplicas tienen los mismos valores para los elementos de datos actualizados (es decir, son mutuamente consistentes), y el historial global resultante es 1SR.

Los dos parámetros de diseño que analizamos anteriormente determinan la implementación específica de los protocolos de réplica centralizada diligente: dónde se realizan las actualizaciones y el grado de transparencia de la replicación. El primer parámetro, descrito en la sección 6.2.3, se refiere a si existe un único sitio maestro para todos los elementos de datos (maestro único), a diferentes sitios maestros para cada uno o, más probablemente, a un grupo de elementos de datos (copia principal). El segundo parámetro indica si cada aplicación conoce la ubicación de la copia maestra (transparencia limitada de la aplicación) o si puede confiar en su TM local para determinarla (transparencia total de la replicación).

6.3.1.1 Maestro único con transparencia de replicación limitada

El caso más simple es tener una única base de datos maestra para toda la base de datos (es decir, para todos los elementos de datos) con transparencia de replicación limitada para que las aplicaciones de usuario conozcan el sitio maestro. En este caso, las transacciones de actualización global (es decir, aquellas que contienen al menos una operación $W(x)$, donde x es un elemento de datos replicado) se envían directamente al sitio maestro, más específicamente, a la TM en el sitio maestro. En el maestro, cada operación $R(x)$ se realiza en la copia maestra (es decir, $R(x)$ se convierte en $R(xM)$, donde M significa copia maestra) y se ejecuta de la siguiente manera: se obtiene un bloqueo de lectura en xM , se realiza la lectura y el resultado se devuelve al usuario. De manera similar, cada $W(x)$ causa una actualización de la copia maestra [es decir, se ejecuta como $W(xM)$] obteniendo primero un bloqueo de escritura y luego realizando la operación de escritura. La TM maestra luego reenvía la escritura a los sitios esclavos ya sea de forma síncrona o diferida (Fig. 6.1).

En cualquier caso, es importante propagar las actualizaciones de forma que las actualizaciones conflictivas se ejecuten en los esclavos en el mismo orden que en el maestro. Esto puede lograrse mediante marcas de tiempo o algún otro esquema de ordenación.

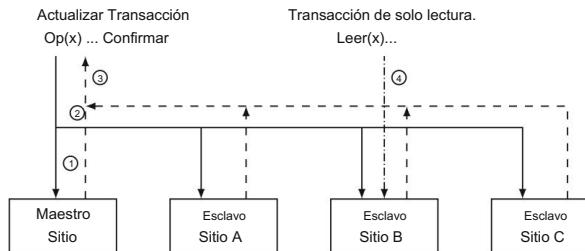


Fig. 6.1 Acciones del protocolo de replicación maestra única Eager. (1) Se aplica una escritura en la copia maestra; (2) Luego, la escritura se propaga a las demás réplicas; (3) Las actualizaciones se vuelven permanentes en el momento de la confirmación; (4) La lectura de la transacción de solo lectura se dirige a cualquier copia esclava.

La aplicación de usuario puede enviar una transacción de solo lectura (es decir, todas las operaciones son de lectura) a cualquier sitio esclavo. La ejecución de transacciones de solo lectura en los esclavos puede seguir el proceso de algoritmos de control de concurrencia centralizados, como C2PL (Algoritmos 5.1–5.3), donde el administrador de bloques centralizado reside en el sitio de réplica maestro. Las implementaciones dentro de C2PL requieren cambios mínimos en el TM en los sitios no maestros, principalmente para lidiar con las operaciones de escritura como se describió anteriormente, y sus consecuencias (por ejemplo, en el procesamiento del comando Commit). Por lo tanto, cuando un sitio esclavo recibe una operación de lectura (de una transacción de solo lectura), la reenvía al sitio maestro para obtener un bloqueo de lectura. La lectura puede luego ejecutarse en el maestro y el resultado devolverse a la aplicación, o el maestro puede simplemente enviar un mensaje de "bloqueo concedido" al sitio de origen, que luego puede ejecutar la lectura en la copia local.

Es posible reducir la carga en el maestro realizando la lectura en la copia local sin obtener un bloqueo de lectura del sitio maestro. Ya sea que se utilice propagación síncrona o diferida, el algoritmo de control de concurrencia local garantiza que los conflictos de lectura-escritura locales se serialicen correctamente. Dado que las operaciones de escritura solo pueden provenir del maestro como parte de la propagación de actualizaciones, no se producirán conflictos de escritura-escritura locales, ya que las transacciones de propagación se ejecutan en cada esclavo en el orden dictado por el maestro. Sin embargo, una lectura puede leer valores de elementos de datos en un esclavo antes o después de la instalación de una actualización. El hecho de que una lectura de una transacción en un sitio esclavo pueda leer el valor de una réplica antes de una actualización, mientras que otra lectura de otra transacción lea otra réplica en otro esclavo después de la misma actualización, es irrelevante para garantizar los históricos globales de 1SR. Esto se demuestra con el siguiente ejemplo.

Ejemplo 6.3 Considere un elemento de datos x cuyo sitio maestro está en el sitio A con esclavos en los sitios B y C. Considere las siguientes tres transacciones:

T1: Escribir(x) T2: Leer(x) T3: Leer(x)

Comprometerse

Comprometerse

Comprometerse

Suponga que T2 se envía al esclavo en el sitio B y T3 al esclavo en el sitio C. Suponga que T2 lee x en B [R2(xB)] antes de que se aplique la actualización de T1 en B, mientras que T3 lee x en C

[R3(xC)] después de la actualización de T1 en C. Entonces los historiales generados en los dos esclavos serán los siguientes:

$$HB = \{R2(x), C2, W1(x), C1\}$$

$$HC = \{W1(x), C1, R3(x), C3\}$$

El orden de serialización en el sitio B es $T2 \rightarrow T1$, mientras que en el sitio C es $T1 \rightarrow T3$. Por lo tanto, el orden de serialización global es $T2 \rightarrow T1 \rightarrow T3$, lo cual es correcto. Por lo tanto, el historial es 1SR.

En consecuencia, si se sigue este enfoque, las transacciones de lectura pueden leer datos que se actualizan simultáneamente en el maestro, pero el historial global seguirá siendo 1SR.

En este protocolo alternativo, cuando un sitio esclavo Si recibe una $R(x)$, obtiene un bloqueo de lectura local, lee de su copia local [es decir, $R(xi)$] y devuelve el resultado a la aplicación del usuario; esto solo puede provenir de una transacción de solo lectura. Cuando recibe una $W(x)$, si proviene del sitio maestro, la ejecuta en la copia local [es decir, $Wi(xi)$]. Si proviene de una aplicación del usuario, rechaza $W(x)$, ya que esto es obviamente un error, dado que las transacciones de actualización deben enviarse al sitio maestro.

Estas alternativas de un protocolo centralizado con un solo maestro son fáciles de implementar. Un aspecto importante a considerar es cómo se reconoce una transacción como de "actualización" o de "solo lectura". Esto podría hacerse mediante una declaración explícita dentro del comando `Begin_transaction`.

6.3.1.2 Maestro único con transparencia de replicación completa

Los protocolos centralizados con un solo maestro requieren que cada aplicación de usuario conozca el sitio maestro, lo que supone una carga significativa para el maestro, que debe gestionar (al menos) las operaciones de lectura dentro de las transacciones de actualización, además de actuar como coordinador de estas transacciones durante la ejecución de 2PC. Estos problemas se pueden solucionar, en cierta medida, involucrando en la ejecución de las transacciones de actualización a la TM del sitio donde se ejecuta la aplicación. De este modo, las transacciones de actualización no se envían al maestro, sino a la TM del sitio donde se ejecuta la aplicación (ya que no necesitan conocer al maestro). Esta TM puede actuar como coordinadora tanto para las transacciones de actualización como para las de solo lectura. Las aplicaciones pueden simplemente enviar sus transacciones a su TM local, lo que proporciona total transparencia.

Existen alternativas para implementar la transparencia total: la TM coordinadora puede actuar únicamente como un "enrutador", reenviando cada operación directamente al sitio maestro. Este puede entonces ejecutar las operaciones localmente (como se describió anteriormente) y devolver los resultados a la aplicación. Si bien esta implementación alternativa proporciona transparencia total y tiene la ventaja de ser sencilla de implementar, no soluciona el problema de sobrecarga en el sitio maestro. Una implementación alternativa podría ser la siguiente:

1. La TM coordinadora envía cada operación, conforme la recibe, al sitio central (maestro). Esto no requiere cambios en el algoritmo C2PL-TM (Algoritmo 5.1).
2. Si la operación es una R(x), el gestor de bloqueos centralizado (C2PL-LM en el algoritmo 5.2) puede proceder estableciendo un bloqueo de lectura en su copia de x (llamada xM) en nombre de esta transacción e informa al TM coordinador que el bloqueo de lectura está concedido. El TM coordinador puede entonces reenviar la R(x) a cualquier sitio esclavo que contenga una réplica de x [es decir, convertirla en una R(xi)]. La lectura puede entonces ser realizada por el procesador de datos (PD) en ese esclavo.
3. Si la operación es una W (x), entonces el administrador de cerradura centralizada (maestro) procede como sigue:
 - (a) Primero establece un bloqueo de escritura en su copia de xM.
 - (b) Luego llama a su DP local para realizar W (xM) en su propia copia.
 - (c) Finalmente, informa al TM coordinador que se concede el bloqueo de escritura.

La TM coordinadora, en este caso, envía la W (x) a todos los esclavos donde existe una copia de x; los DP en estos esclavos aplican la Escritura a sus copias locales.

La diferencia fundamental en este caso radica en que el sitio maestro no se encarga de la lectura ni de la coordinación de las actualizaciones entre réplicas. Estas tareas quedan en manos de la TM en el sitio donde se ejecuta la aplicación del usuario.

Es evidente que este algoritmo garantiza que los históricos sean 1SR, ya que los órdenes de serialización se determinan en un único maestro (similar a los algoritmos de control de concurrencia centralizados). También es evidente que el algoritmo sigue el protocolo ROWA, como se mencionó anteriormente: dado que se garantiza que todas las copias estén actualizadas al completarse una transacción de actualización, se puede realizar una lectura en cualquier copia.

Para demostrar cómo los algoritmos entusiastas combinan el control de réplicas y el control de concurrencia, mostramos el algoritmo de Gestión de Transacciones para la TM coordinadora (Algoritmo 6.1) y el algoritmo de Gestión de Bloqueos para el sitio maestro (Algoritmo 6.2). Mostramos únicamente las revisiones de los algoritmos 2PL centralizados (Algoritmos 5.1 y 5.2 en el Capítulo 5).

Tenga en cuenta que en los fragmentos de algoritmo que hemos proporcionado, el LM simplemente envía un mensaje de "Bloqueo concedido" y no el resultado de la operación de actualización. En consecuencia, cuando la TM coordinadora reenvía la actualización a los esclavos, estos deben ejecutar la operación de actualización ellos mismos. Esto a veces se denomina transferencia de operación. La alternativa es que el mensaje "Bloqueo concedido" incluya el resultado del cálculo de actualización, que luego se reenvía a los esclavos, quienes simplemente deben aplicar el resultado y actualizar sus registros. Esto se denomina transferencia de estado. La distinción puede parecer trivial si las operaciones se presentan simplemente en la forma W(x), pero recuerde que esta operación de escritura es una abstracción; cada operación de actualización puede requerir la ejecución de una expresión SQL, en cuyo caso la distinción es muy importante.

La implementación del protocolo descrita anteriormente alivia parte de la carga en el sitio maestro y reduce la necesidad de que las aplicaciones de usuario lo conozcan. Sin embargo, su implementación es más compleja que la primera alternativa que analizamos. En

Algoritmo 6.1: Modificaciones de Eager Single Master a C2PL-TM

```

comenzar
  :
  :
  Si se concede la solicitud de bloqueo,
    entonces si op.Type = W
      | entonces S ← conjunto de todos los sitios que son esclavos del elemento de datos
      demás
      | S ← cualquier sitio que tenga una copia del elemento de
      datos final si
      DPS (op)                                {enviar operación a todos los sitios del conjunto S}
      de
      | lo contrario informará al usuario sobre la terminación de la
      transacción si
      :
    fin
  :

```

Algoritmo 6.2: Modificaciones de Eager Single Master a C2PL-LM

```

comenzar
  :
  :
  interruptor op.Type do
    caso R o W do
      | {solicitud de bloqueo; ver si se puede conceder}
      | encuentra la unidad de bloqueo lu tal que op.arg
      | lu ; si lu está desbloqueado o el modo de bloqueo de lu es compatible con op.Type entonces
      | establecer bloqueo en lu en modo apropiado en nombre de la transacción op.t id; si
      | op.Type = W entonces
      |   | DPM (op)          {llamar al DP local (M para "maestro") con operación}
      |   envía "Bloqueo concedido" al TM coordinador de la transacción; de
      |   lo
      |   | contrario, coloca op en una cola
      |   para lu end if
    caso final
    :
  interruptor final
fin

```

En particular, ahora la TM del sitio donde se envían las transacciones debe actuar como coordinador de 2PC y el sitio maestro se convierte en participante. Esto requiere cierta cautela al revisar los algoritmos en estos sitios.

6.3.1.3 Copia principal con transparencia de replicación completa

Ahora, flexibilicemos el requisito de que exista un único maestro para todos los elementos de datos; cada elemento de datos puede tener un maestro diferente. En este caso, para cada elemento de datos replicado, una de las réplicas se designa como copia principal. Por consiguiente, no existe un único...

El maestro determina el orden de serialización global, por lo que se requiere mayor cuidado. En el caso de bases de datos con replicación completa, cualquier réplica puede ser la copia principal de un elemento de datos; sin embargo, para bases de datos con replicación parcial, la opción de transparencia de replicación limitada solo tiene sentido si una transacción de actualización accede únicamente a elementos de datos cuyos sitios principales se encuentran en el mismo sitio. De lo contrario, el programa de aplicación no puede reenviar las transacciones de actualización a un maestro; tendrá que hacerlo operación por operación y, además, no está claro qué maestro de la copia principal serviría como coordinador para la ejecución de 2PC. Por lo tanto, la alternativa razonable es la transparencia total, donde la TM en el sitio de aplicación actúa como la TM coordinadora y reenvía cada operación al sitio principal del elemento de datos sobre el que actúa. La Figura 6.2 muestra la secuencia de operaciones en este caso, donde flexibilizamos nuestra suposición previa de replicación completa. El sitio A es el maestro del elemento de datos x y los sitios B y C contienen réplicas (es decir, son esclavos); de forma similar, el maestro del elemento de datos y es el sitio C, con los sitios esclavos B y D.

Recuerde que esta versión aún aplica las actualizaciones a todas las réplicas dentro de los límites transaccionales, lo que requiere la integración con técnicas de control de concurrencia. Una propuesta muy temprana es el algoritmo de bloqueo de dos fases de copia primaria (PC2PL), propuesto para la versión distribuida prototípica de INGRES. PC2PL es una extensión directa del protocolo maestro único mencionado anteriormente, en un intento por contrarrestar los posibles problemas de rendimiento de este último. Básicamente, implementa administradores de bloqueos en varios sitios y responsabiliza a cada administrador de bloqueos de la gestión de los bloqueos de un conjunto determinado de unidades de bloqueo del cual es el sitio maestro. Los administradores de transacciones envían entonces sus solicitudes de bloqueo y desbloqueo a los administradores de bloqueos responsables de esa unidad de bloqueo específica. Por lo tanto, el algoritmo trata una copia de cada elemento de datos como su copia primaria.

Como técnica combinada de control de réplica/control de concurrencia, el enfoque de copia primaria exige un directorio más sofisticado en cada sitio, pero también mejora los enfoques discutidos anteriormente al reducir la carga del sitio maestro sin causar una gran cantidad de comunicación entre los administradores de transacciones y los administradores de bloqueo.

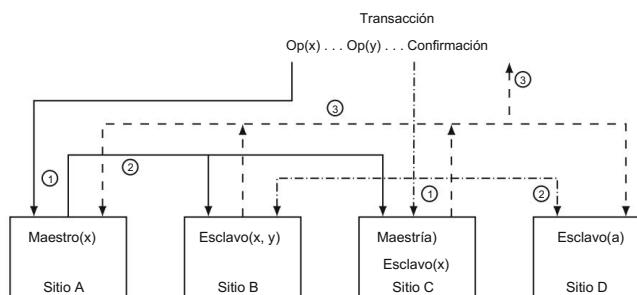


Fig. 6.2 Acciones del protocolo de replicación de copia primaria ansiosa. (1) Las operaciones (lectura o escritura) para cada elemento de datos se enrutan al maestro de ese elemento de datos y primero se aplica una escritura en el maestro; (2) Luego, la escritura se propaga a las demás réplicas; (3) Las actualizaciones se vuelven permanentes en el momento de la confirmación.

6.3.2 Protocolos distribuidos Eager

En el control de réplicas distribuidas entusiastas, las actualizaciones pueden originarse en cualquier lugar y primero se aplican en la réplica local, para luego propagarse a otras réplicas.

Si la actualización se origina en un sitio donde no existe una réplica del elemento de datos, se reenvía a uno de los sitios de réplica, que coordina su ejecución. Nuevamente, todo esto se realiza en el contexto de la transacción de actualización, y cuando esta se confirma, se notifica al usuario y las actualizaciones se hacen permanentes. La Figura 6.3 muestra la secuencia de operaciones para un elemento de datos lógico x con copias en los sitios A, B, C y D, y donde dos transacciones actualizan dos copias diferentes (en los sitios A y D).

Como se puede observar claramente, la cuestión crucial es garantizar que las operaciones de escritura conflictivas concurrentes, iniciadas en diferentes sitios, se ejecuten en el mismo orden en todos los sitios donde se ejecutan conjuntamente (por supuesto, las ejecuciones locales en cada sitio también deben ser serializables). Esto se logra mediante las técnicas de control de concurrencia empleadas en cada sitio. En consecuencia, las operaciones de lectura se pueden realizar en cualquier copia, pero las escrituras se realizan en todas las copias dentro de los límites transaccionales (por ejemplo, ROWA) mediante un protocolo de control de concurrencia.

6.3.3 Protocolos centralizados perezosos

Los algoritmos de replicación centralizada perezosa son similares a los de replicación centralizada diligente en que las actualizaciones se aplican primero a una réplica maestra y luego se propagan a las esclavas. La diferencia importante radica en que la propagación no ocurre dentro de la transacción de actualización, sino después de que esta se confirme como una transacción de actualización independiente. Por lo tanto, si un sitio esclavo realiza una operación $R(x)$ en su copia local, podría leer datos obsoletos (no actualizados), ya que x puede haberse actualizado en la réplica maestra, pero la actualización podría no haberse propagado aún a las esclavas.

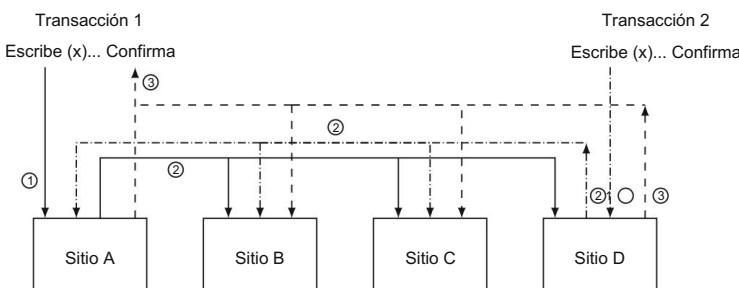


Fig. 6.3 Acciones del protocolo de replicación distribuida Eager. (1) Se aplican dos operaciones de escritura a dos réplicas locales del mismo elemento de datos; (2) Las operaciones de escritura se propagan independientemente a las demás réplicas; (3) Las actualizaciones se vuelven permanentes al momento de la confirmación (mostrado solo para la Transacción 1).

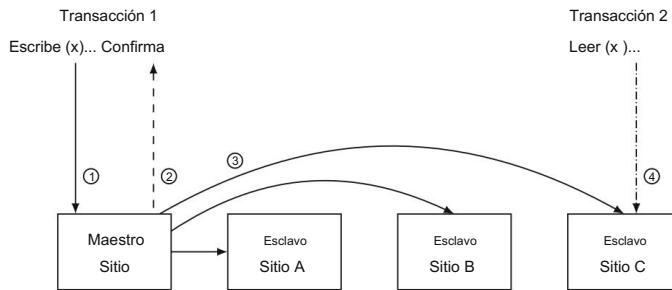


Fig. 6.4 Acciones del protocolo de replicación maestra única perezosa. (1) La actualización se aplica en la réplica local; (2) La confirmación de la transacción hace que las actualizaciones sean permanentes en la maestra; (3) La actualización se propaga a las demás réplicas en transacciones de actualización; (4) La transacción 2 lee desde la copia local.

6.3.3.1 Maestro único con transparencia limitada

En este caso, las transacciones de actualización se envían y ejecutan directamente en el sitio maestro (como en el maestro único ansioso); una vez confirmada la transacción de actualización, la transacción de actualización se envía a los esclavos. La secuencia de pasos de ejecución es la siguiente: (1) primero se aplica una transacción de actualización a la réplica del maestro, (2) la transacción se confirma en el maestro y, a continuación, (3) la transacción de actualización se envía a los esclavos (Fig. 6.4).

Cuando un sitio esclavo recibe una $R(x)$, lee de su copia local y devuelve el resultado al usuario. Tenga en cuenta que, como se indicó anteriormente, su propia copia podría no estar actualizada si el sitio maestro se está actualizando y el esclavo aún no ha recibido ni ejecutado la transacción de actualización correspondiente. Una $R(x)$ recibida por un esclavo se rechaza (y la transacción se cancela), ya que debería haberse enviado directamente al sitio maestro.

Cuando un esclavo recibe una transacción de actualización del maestro, aplica las actualizaciones a su copia local. Cuando recibe una confirmación o una cancelación (la cancelación solo puede ocurrir para transacciones de solo lectura enviadas localmente), realiza estas acciones localmente.

El caso de la copia primaria con transparencia limitada es similar, por lo que no lo analizaremos en detalle. En lugar de ir a un único sitio maestro, $W(x)$ se envía a la copia primaria de x ; el resto es sencillo.

¿Cómo se puede garantizar que las transacciones de actualización se apliquen en todos los esclavos en el mismo orden? En esta arquitectura, dado que existe una única copia maestra para todos los elementos de datos, el orden se puede establecer simplemente mediante marcas de tiempo. El sitio maestro asignaría una marca de tiempo a cada transacción de actualización según el orden de confirmación de la transacción de actualización real, y los esclavos aplicarían las transacciones de actualización en el orden de la marca de tiempo.

Se puede seguir un enfoque similar en el caso de la copia principal con transparencia limitada. En este caso, un sitio contiene copias esclavas de varios elementos de datos, lo que provoca que reciba transacciones de actualización de varios maestros. La ejecución de estas transacciones de actualización debe ordenarse de la misma manera en todos los esclavos involucrados para garantizar...

Que los estados de la base de datos eventualmente converjan. Existen varias alternativas.

Una alternativa es asignar marcas de tiempo de manera que las transacciones de actualización emitidas desde diferentes maestros tengan diferentes marcas de tiempo (añadiendo el identificador del sitio a un contador monótono en cada sitio). Entonces, las transacciones de actualización en cada sitio pueden ejecutarse en el orden de su marca de tiempo. Sin embargo, las que salen fuera de orden causan dificultades. En las técnicas tradicionales basadas en marcas de tiempo discutidas en el Cap. 5, estas transacciones se abortarían; sin embargo, en la replicación perezosa, esto no es posible ya que la transacción ya se ha confirmado en el sitio de copia principal. La única posibilidad es ejecutar una transacción de compensación (que, efectivamente, aborta la transacción revirtiendo sus efectos) o realizar una conciliación de actualizaciones que se discutirá en breve. El problema puede abordarse mediante un estudio más cuidadoso de los historiales resultantes. Un enfoque es utilizar un enfoque de grafo de serialización que construye un grafo de replicación cuyos nodos consisten en transacciones (T) y sitios (S) y una arista T_i, S_j existe en el grafo si y solo si T_i realiza una escritura en una copia física (replicada) que está almacenada en S_j . Cuando se envía una operación (op_k), los nodos (T_k) y las aristas correspondientes se insertan en el grafo de replicación, donde se verifican los ciclos. Si no hay ciclo, la ejecución puede continuar. Si se detecta un ciclo e involucra una transacción confirmada en el maestro, pero cuyas transacciones de actualización aún no se han confirmado en ninguno de los esclavos involucrados, la transacción actual (T_k) se cancela (para reiniciarse posteriormente), ya que su ejecución provocaría que el historial no sea de primera respuesta (1SR). De lo contrario, T_k puede esperar hasta que se completen las demás transacciones del ciclo (es decir, que se confirmen en sus maestros y sus transacciones de actualización en todos los esclavos). Cuando una transacción se completa de esta manera, el nodo correspondiente y todas sus aristas incidentes se eliminan del grafo de replicación. Se ha comprobado que este protocolo genera historiales de primera respuesta (1SR). Un aspecto importante es el mantenimiento del grafo de replicación. Si lo mantiene un solo sitio, se convierte en un algoritmo centralizado. Dejamos la construcción y el mantenimiento distribuidos del grafo de replicación como ejercicio.

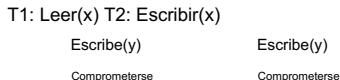
Otra alternativa es confiar en el mecanismo de comunicación grupal proporcionado por la infraestructura de comunicación subyacente (si esta puede proporcionarlo). Analizamos esta alternativa en la sección 6.4.

Recordemos de la Sección 6.3.1 que, en el caso de bases de datos parcialmente replicadas, el enfoque de copia primaria diligente con transparencia de replicación limitada tiene sentido si las transacciones de actualización acceden únicamente a elementos de datos cuyos sitios maestros son los mismos, ya que las transacciones de actualización se ejecutan completamente en un maestro. El mismo problema existe en el caso de la copia primaria perezosa con enfoque de replicación limitada. La cuestión que surge en ambos casos es cómo diseñar la base de datos distribuida para que se puedan ejecutar transacciones significativas. Este problema se ha estudiado en el contexto de los protocolos perezosos y se propuso un algoritmo de selección de sitio primario que, dado un conjunto de transacciones, un conjunto de sitios y un conjunto de elementos de datos, encuentra una asignación de sitio primario a estos elementos de datos (si existe) de modo que el conjunto de transacciones pueda ejecutarse para generar un historial global de 1SR.

6.3.3.2 Copia maestra única o primaria con transparencia de replicación total

Ahora analizamos alternativas que ofrecen total transparencia, permitiendo el envío de transacciones (tanto de lectura como de actualización) en cualquier sitio y el reenvío de sus operaciones al maestro único o al sitio maestro principal correspondiente. Esto es complejo y conlleva dos problemas: el primero es que, a menos que se tenga cuidado, el historial global de 1SR podría no estar garantizado; el segundo es que una transacción podría no ver sus propias actualizaciones. Los dos ejemplos siguientes ilustran estos problemas.

Ejemplo 6.4 Considere el escenario de un solo maestro y dos sitios M y B, donde M contiene las copias maestras de x e y, y B contiene sus copias esclavas. Considere ahora las dos transacciones siguientes: T1 enviada al sitio B, mientras que la transacción T2 se envía al sitio M:



Una forma de ejecutar esto con total transparencia es la siguiente. T2 se ejecutaría en el sitio M, ya que contiene las copias maestras de x e y. Poco después de confirmar, se envían transacciones de actualización para sus operaciones de escritura al sitio B para actualizar las copias esclavas. Por otro lado, T1 leería la copia local de x en el sitio B [R1(xB)], pero su W1(x) se reenviaría a la copia maestra de x, que se encuentra en el sitio M. Poco después de que W1(x) se ejecute en el sitio maestro y se confirme allí, se enviaría una transacción de actualización al sitio B para actualizar la copia esclava. La siguiente es una posible secuencia de pasos de ejecución (Fig. 6.5):

1. R1(x) se envía en el sitio B, donde se realiza [R1(xB)]; 2. W2(x) se envía en el sitio M y se ejecuta [W2(xM)]; 3. W2(y) se envía en el sitio M y se ejecuta [W2(yB)]; 4. T2 envía su confirmación en el sitio M y se confirma allí; 5. W1(x) se envía en el sitio B; dado que la copia maestra de x está en el sitio M, la escritura es remitido a M;
6. W1(x) se ejecuta en el sitio M [W1(xM)]; y la confirmación se envía de vuelta al sitio B; 7. T1 envía una confirmación en el sitio B, que la reenvía al sitio M; se ejecuta allí y se informa a B de la confirmación donde T1 también confirma; 8. El sitio M ahora envía una transacción de actualización para T2 al sitio B donde se ejecuta y se compromete;
9. El sitio M finalmente envía una transacción de actualización para T1 al sitio B (esto es para la escritura de T1 que se ejecutó en el maestro), se ejecuta en B y se confirma.

Las dos historias siguientes se generan ahora en los dos sitios donde se encuentran
El superíndice r en las operaciones indica que son parte de una transacción de actualización:

$$HB = \{W2(xM), W2(yM), C2, W1(yM), C1\}$$

$$HB = \{R1(xB), C1, Wr(xB), Wr(yB), Cr_2, Wr_1(xB), Cr_1\}$$

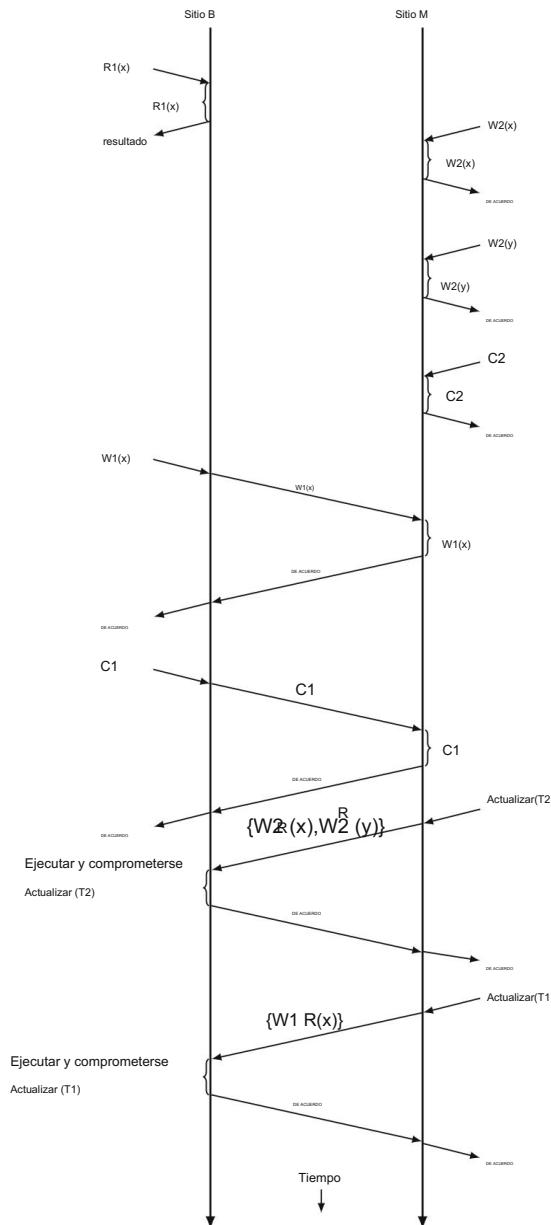


Fig. 6.5 Secuencia temporal de ejecuciones de transacciones

El historial global resultante sobre los elementos de datos lógicos x e y no es 1SR.

Ejemplo 6.5 Consideremos nuevamente un escenario con un solo maestro, donde el sitio M contiene la copia maestra de x y el sitio D contiene su esclavo. Consideremos la siguiente transacción simple:

T3: Escribir(x)

Leer(x)

Comprometerse

Siguiendo el mismo modelo de ejecución que en el Ejemplo 6.4, la secuencia de pasos sería como sigue:

1. W3(x) se envía en el sitio D, que lo reenvía al sitio M para su ejecución; 2. La escritura se ejecuta en M [W3(xM)] y la confirmación se envía de vuelta al sitio D; 3. R3(x) se envía en el sitio D y se ejecuta en la copia local [R3(xD)]; 4. T3 envía la confirmación en D, que se reenvía a M, se ejecuta allí y se envía una notificación de vuelta al sitio D, que también confirma la transacción;
5. El sitio M envía una transacción de actualización al sitio D para la operación W3(x);
6. El sitio D ejecuta la transacción de actualización y la confirma.

Tenga en cuenta que, dado que la transacción de actualización se envía al sitio D en algún momento después de que T3 se confirma en el sitio M, en el paso 3 cuando lee el valor de x en el sitio D, lee el valor anterior y no ve el valor de su propia escritura que precede a la lectura.

Debido a estos problemas, no hay demasiadas propuestas para la transparencia total en algoritmos de replicación perezosa. Una excepción notable es un algoritmo que considera el caso de un solo maestro y proporciona un método para la prueba de validez por parte del sitio maestro, en el punto de confirmación, similar al control de concurrencia optimista. La idea fundamental es la siguiente. Considere una transacción T que escribe un elemento de datos x. En el momento de confirmación de la transacción T, el maestro genera una marca de tiempo para él y usa esta marca de tiempo para establecer una marca de tiempo para la copia maestra (xM) que registra la marca de tiempo de la última transacción que lo actualizó (`last_modif ied(xM)`). Esto también se adjunta a las transacciones de actualización. Cuando se reciben transacciones de actualización en los esclavos, también establecen sus copias en este mismo valor, es decir, $\text{last_modif ied}(xi) \leftarrow \text{last_modif ied}(xM)$.

La generación de la marca de tiempo para T en el maestro sigue la siguiente regla:

La marca de tiempo de la transacción T debe ser mayor que todas las marcas de tiempo emitidas previamente y menor que las marcas de tiempo `last_modif ied` de los elementos de datos a los que ha accedido. Si no se puede generar dicha marca de tiempo, la transacción T se cancela.³

Esta prueba garantiza que las operaciones de lectura lean los valores correctos. Por ejemplo, en el Ejemplo 6.4, el sitio maestro M no podría asignar una marca de tiempo adecuada.

³La propuesta original gestiona una amplia gama de restricciones de frescura, como ya se mencionó; por lo tanto, la regla se especifica de forma más genérica. Sin embargo, dado que nuestro análisis se centra principalmente en el comportamiento de 1SR, esta reformulación (más estricta) de la regla resulta adecuada.

A la transacción T1 al confirmarse, ya que el `last_modif ied(xM)` reflejaría la actualización realizada por T2. Por lo tanto, T1 se cancelaría.

Aunque este algoritmo soluciona el primer problema que mencionamos anteriormente, no soluciona automáticamente el problema de que una transacción no vea sus propias escrituras (lo que denominamos inversión de transacción anteriormente). Para solucionar este problema, se ha sugerido mantener una lista de todas las actualizaciones que realiza una transacción, la cual se consulta al ejecutar una lectura. Sin embargo, dado que solo el maestro conoce las actualizaciones, la lista debe mantenerse en el maestro y todas las operaciones de lectura y escritura deben ejecutarse en él.

6.3.4 Protocolos distribuidos perezosos

Los protocolos de replicación distribuida perezosa son los más complejos debido al hecho de que las actualizaciones pueden ocurrir en cualquier réplica y se propagan a las otras réplicas de forma perezosa (Fig. 6.6).

El funcionamiento del protocolo en el sitio donde se envía la transacción es sencillo: tanto las operaciones de lectura como las de escritura se ejecutan en la copia local, y la transacción se confirma localmente. Poco después de la confirmación, las actualizaciones se propagan a los demás sitios mediante transacciones de actualización.

Las complicaciones surgen al procesar estas actualizaciones en los demás sitios. Cuando las transacciones de actualización llegan a un sitio, deben programarse localmente, lo cual se realiza mediante el mecanismo de control de concurrencia local. La serialización correcta de estas transacciones de actualización puede lograrse mediante las técnicas descritas en secciones anteriores. Sin embargo, varias transacciones pueden actualizar diferentes copias del mismo dato simultáneamente en distintos sitios, y estas actualizaciones pueden entrar en conflicto entre sí. Estos cambios deben conciliarse y esto complica el orden de actualización.

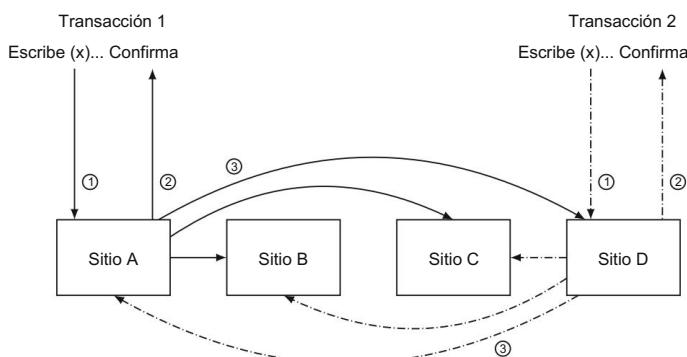


Fig. 6.6 Acciones del protocolo de replicación distribuida diferida. (1) Se aplican dos actualizaciones en dos réplicas locales; (2) La confirmación de la transacción hace que las actualizaciones sean permanentes; (3) Las actualizaciones se propagan independientemente a las demás réplicas.

Transacciones. Con base en los resultados de la conciliación, se determina el orden de ejecución de las transacciones de actualización y se aplican las actualizaciones en cada sitio.

El problema crítico aquí es la conciliación. Se puede diseñar un algoritmo de conciliación de propósito general basado en heurística. Por ejemplo, las actualizaciones se pueden aplicar en orden de marca de tiempo (es decir, las que tengan marcas de tiempo posteriores siempre prevalecerán) o se puede dar preferencia a las actualizaciones que se originan en ciertos sitios (quizás haya sitios más importantes). Sin embargo, estos son métodos ad hoc y la conciliación depende realmente de la semántica de la aplicación. Además, independientemente de la técnica de conciliación utilizada, algunas actualizaciones se pierden. Tenga en cuenta que el ordenamiento basado en marcas de tiempo solo funcionará si estas se basan en relojes locales sincronizados. Como se mencionó anteriormente, esto es difícil de lograr en sistemas distribuidos a gran escala. Un enfoque simple basado en marcas de tiempo, que concatena un número de sitio y un reloj local, otorga preferencia arbitraria entre transacciones que pueden no tener una base real en la lógica de la aplicación. La razón por la que las marcas de tiempo funcionan bien en el control de concurrencia y no en este caso es porque en el control de concurrencia solo nos interesa determinar un orden; aquí nos interesa determinar un orden particular que sea consistente con la semántica de la aplicación.

6.4 Comunicación grupal

Como se explicó en la sección anterior, la sobrecarga de los protocolos de replicación puede ser elevada, especialmente en términos de sobrecarga de mensajes. Un modelo de costos muy simple para los algoritmos de replicación es el siguiente: si hay n réplicas y cada transacción consta de m operaciones de actualización, cada transacción emite $n \times m$ mensajes (si la comunicación multidifusión es posible, m mensajes serían suficientes). Si el sistema desea mantener un rendimiento de k transacciones por segundo, esto resulta en $k \times n \times m$ mensajes por segundo (o $k \times m$ en el caso de la multidifusión). Se puede sofisticar esta función de costos considerando el tiempo de ejecución de cada operación (quizás en función de la carga del sistema) para obtener una función de costos en términos de tiempo.

El problema con muchos de los protocolos de replicación analizados anteriormente (en particular los distribuidos) es que su sobrecarga de mensajes es alta.

Un aspecto crítico para la implementación eficiente de estos protocolos es la reducción de la sobrecarga de mensajes. Se han propuesto soluciones que utilizan protocolos de comunicación grupal junto con técnicas no tradicionales para procesar transacciones locales.

Estas soluciones introducen dos modificaciones: no emplean 2PC en el momento de la confirmación, sino que se basan en los protocolos de comunicación grupal subyacentes para garantizar el acuerdo, y utilizan propagación de actualización diferida en lugar de sincrónica.

Repasemos primero el concepto de comunicación grupal. Un sistema de comunicación grupal permite a un nodo realizar la multidifusión de un mensaje a todos los nodos de un grupo con garantía de entrega; es decir, el mensaje se entrega finalmente a todos los nodos. Además, puede proporcionar primitivas de multidifusión con diferentes órdenes de entrega, de los cuales solo uno es importante para nuestro análisis: el orden total. En la multidifusión con orden total, todos los mensajes...

Los datos enviados por diferentes nodos se entregan en el mismo orden total a todos los nodos. Esto es importante para comprender la siguiente explicación.

Demostraremos el uso de la comunicación grupal considerando dos protocolos. El primero es un protocolo distribuido alternativo y entusiasta, mientras que el segundo es un protocolo centralizado y perezoso.

El protocolo distribuido ansioso basado en comunicación grupal utiliza una estrategia de procesamiento local donde las operaciones de escritura se llevan a cabo en copias de sombra locales donde se envía la transacción y utiliza una comunicación de grupo totalmente ordenada para difundir el conjunto de operaciones de escritura de la transacción a todos los demás sitios de réplica.

La comunicación totalmente ordenada garantiza que todos los sitios reciban las operaciones de escritura en el mismo orden, asegurando así un orden de serialización idéntico en cada sitio. Para simplificar la explicación, en la siguiente discusión, asumimos que la base de datos está completamente replicada y que cada sitio implementa un algoritmo de control de concurrencia 2PL.

El protocolo ejecuta una transacción T_i en cuatro pasos (control de concurrencia local)
Las acciones no están indicadas):

- I. Fase de procesamiento local. Se realiza una operación $R_i(x)$ en el sitio donde se envía (este es el sitio maestro para esta transacción). También se realiza una operación $W_i(x)$ en el sitio maestro, pero en una instantánea (véase el capítulo anterior para una explicación sobre la paginación instantánea).
- II. Fase de comunicación. Si T_i consta solo de operaciones de lectura, se puede confirmar en el sitio maestro. Si implica operaciones de escritura (es decir, si se trata de una transacción de actualización), la TM en el sitio maestro de T_i (es decir, el sitio donde se envía T_i)⁴ y la multidifunde a todos los sitios de réplica de WMi (incluido él mismo) utilizando una comunicación de grupo totalmente ordenada.
- III. Fase de bloqueo. Cuando WMi se entrega en un sitio S_j , solicita todos los bloqueos de WMi en un paso atómico. Esto se puede lograr adquiriendo un latch (una forma más ligera de bloqueo) en la tabla de bloqueos, que se mantiene hasta que se otorgan todos los bloqueos o se ponen en cola las solicitudes. Se realizan las siguientes acciones:

1. Para cada $W(x)$ en WMi (sea x_j la copia de x que existe en el sitio S_j),
Se realizan los siguientes:

- (a) Si no hay otras transacciones que hayan bloqueado x_j , entonces se concede el bloqueo de escritura en x_j . (b)

De lo contrario, se realiza una prueba de conflicto:

- Si hay una transacción local T_k que ya ha bloqueado x_j , pero está en sus fases de lectura o comunicación local, entonces T_k se cancela. Además, si T_k se encuentra en fase de comunicación, se envía un mensaje de decisión final de "abortar" a todos los sitios. En esta etapa, se detectan conflictos de lectura/escritura y las transacciones de lectura locales simplemente se abortan. Tenga en cuenta que solo las operaciones de lectura locales obtienen bloqueos durante la lectura local.

⁴Lo que se envía son los elementos de datos actualizados (es decir, transferencia de estado).

Fase de ejecución, ya que las escrituras locales solo se ejecutan en instantáneas. Por lo tanto, no es necesario verificar si hay conflictos de escritura en esta etapa. • De lo contrario, la solicitud de bloqueo $W_i(x_j)$ se pone en cola para x_j .

2. Si T_i es una transacción local (recuerde que el mensaje también se envía al sitio donde se origina T_i , en cuyo caso $j = i$), el sitio puede confirmar la transacción, por lo que envía un mensaje de confirmación por multidifusión. Tenga en cuenta que el mensaje de confirmación se envía en cuanto se solicitan los bloqueos y no después de las escrituras; por lo tanto, no se trata de una ejecución 2PC.

IV. Fase de escritura. Cuando un sitio obtiene el bloqueo de escritura, aplica la actualización correspondiente (para el sitio maestro, esto significa que la instantánea se convierte en la versión válida). El sitio donde se envía T_i puede confirmar y liberar todos los bloqueos. Los demás sitios deben esperar el mensaje de decisión y finalizar la sesión en consecuencia.

Tenga en cuenta que en este protocolo, lo importante es garantizar que las fases de bloqueo de las transacciones concurrentes se ejecuten en el mismo orden en cada sitio; esto es lo que se logra con la multidifusión totalmente ordenada. También tenga en cuenta que no existe un requisito de orden para los mensajes de decisión (paso III.2) y que estos pueden entregarse en cualquier orden, incluso antes de la entrega del mensaje de tiempo de respuesta correspondiente. En tal caso, los sitios que reciben el mensaje de decisión antes del mensaje de tiempo de respuesta simplemente registran la decisión, pero no realizan ninguna acción. Cuando llega el mensaje de tiempo de respuesta, pueden ejecutar las fases de bloqueo y escritura y finalizar la transacción según el mensaje de decisión entregado previamente.

Este protocolo es significativamente mejor, en términos de rendimiento, que el protocolo ingenuo descrito en la sección 6.3.2. Para cada transacción, el sitio maestro envía dos mensajes: uno al enviar el WM y el segundo al comunicar la decisión.

Por lo tanto, si deseamos mantener un rendimiento del sistema de k transacciones por segundo, el número total de mensajes es $2k$ en lugar de $k \cdot m$, como ocurre con el protocolo ingenuo (suponiendo multidifusión en ambos casos). Además, el rendimiento del sistema mejora mediante el uso de la propagación diligente diferida, ya que la sincronización entre los sitios de réplica para todas las operaciones de escritura se realiza una sola vez al final, en lugar de durante la ejecución de la transacción.

El segundo ejemplo del uso de la comunicación grupal que analizaremos se da en el contexto de algoritmos centralizados perezosos. Cabe recordar que un aspecto importante en este caso es garantizar que las transacciones de actualización se ordenen de la misma manera en todos los esclavos involucrados, de modo que los estados de la base de datos converjan. Si se dispone de multidifusión totalmente ordenada, las transacciones de actualización enviadas por diferentes sitios maestros se entregarían en el mismo orden en todos los esclavos. Sin embargo, la multidifusión totalmente ordenada tiene una alta sobrecarga de mensajería, lo que puede limitar su escalabilidad. Es posible flexibilizar el requisito de ordenamiento del sistema de comunicación y dejar que el protocolo de replicación se encargue de ordenar la ejecución de las transacciones de actualización. Demostraremos esta alternativa mediante un protocolo que asume la comunicación de multidifusión ordenada FIFO con un retardo de comunicación limitado (llamado Max).

y asume que los relojes están sincronizados de forma vaga, de modo que sólo pueden ser fuera de sincronización hasta Se supone además que hay una transacción apropiada Funcionalidad de gestión en cada sitio. El resultado del protocolo de replicación en Cada esclavo debe mantener una "cola de ejecución" que contiene una lista ordenada de actualizaciones. transacciones, que es la entrada al gestor de transacciones para su ejecución local. Por lo tanto, El protocolo garantiza que los pedidos en las colas en ejecución en cada sitio esclavo donde se encuentra un El conjunto de transacciones de actualización que se ejecutan es el mismo.

En cada sitio esclavo, se mantiene una "cola pendiente" para cada sitio maestro de este esclavo (es decir, si el sitio esclavo tiene réplicas de x e y cuyos sitios maestros son S1 y S2, respectivamente, entonces hay dos colas pendientes, q1 y q2, correspondientes al maestro sitios S1 y S2, respectivamente). Cuando se actualiza una transacción RT del $\overset{k}{i}$ se crea en un master sitio sitek, se le asigna una marca de tiempo ts(RTi) que corresponde al valor en tiempo real. En el momento de confirmación de la transacción de actualización correspondiente Ti. Cuando RTi llega En un esclavo, se pone en la cola qk. Con cada llegada de mensaje, los elementos superiores de todos... Se escanean las colas pendientes y se elige la que tiene la marca de tiempo más baja. El nuevo RT (new_RT) que se gestionará. Si el new_RT ha cambiado desde la última ciclo (es decir, un nuevo RT llegó con una marca de tiempo inferior a la elegida en el ciclo anterior), entonces el que tiene la marca de tiempo más baja se convierte en el new_RT y se considera para la programación.

Cuando se elige una transacción de actualización como new_RT , No se pone inmediatamente en la "cola de ejecución" del administrador de transacciones; la programación de una actualización La transacción tiene en cuenta el retraso máximo y la posible deriva en el mercado local. relojes. Esto se hace para garantizar que cualquier transacción de actualización que pueda retrasarse tenga un Probabilidad de alcanzar al esclavo. El momento en que un RTi se coloca en la cola de ejecución. En un sitio esclavo, el tiempo de entrega es = ts(new_RT)+Max +. Dado que el sistema de comunicación garantiza un límite superior de Max para la entrega de mensajes y dado que La deriva máxima en los relojes locales (que determinan las marcas de tiempo) es una transacción de actualización no puede retrasarse más del tiempo de entrega antes de llegar a todos los esclavos previstos. Por lo tanto, el protocolo garantiza que se programe una transacción de actualización para su ejecución en un esclavo cuando se cumple lo siguiente: (1) todas las operaciones de escritura de Las transacciones de actualización correspondientes se realizan en el maestro, (2) de acuerdo con el orden determinado por la marca de tiempo de la transacción de actualización (que refleja la orden de confirmación de la transacción de actualización), y (3) lo antes posible en tiempo real equivalente a su tiempo de entrega. Esto garantiza que las actualizaciones en las copias secundarias en el Los sitios de esclavos siguen el mismo orden cronológico en el que se encontraron sus copias primarias. actualizado y este orden será el mismo en todos los esclavos involucrados, asumiendo que La infraestructura de comunicación subyacente puede garantizar Max y Esto es un ejemplo de un algoritmo perezoso que garantiza un historial global 1SR, pero una consistencia mutua débil, lo que permite que los valores de la réplica diverjan hasta un período de tiempo predeterminado.

6.5 Replicación y fallos

Hasta este punto, nos hemos centrado en los protocolos de replicación en ausencia de cualquier Fallos. ¿Qué sucede con los problemas de consistencia mutua si se producen fallos del sistema?

El manejo de fallas difiere entre los enfoques de replicación ansiosa y replicación perezosa.

6.5.1 Fallos y replicación diferida

Consideremos primero cómo las técnicas de replicación diferida gestionan los fallos. Este caso es relativamente sencillo, ya que estos protocolos permiten la divergencia entre las copias maestras y las réplicas. Por consiguiente, cuando los fallos de comunicación impiden el acceso a uno o más sitios (debido a la partición de la red), los sitios disponibles pueden simplemente continuar el procesamiento. Incluso con la partición de la red, se pueden permitir operaciones en múltiples particiones de forma independiente y, posteriormente, preocuparse por la convergencia de los estados de la base de datos tras la reparación, utilizando las técnicas de resolución de conflictos descritas en la sección 6.3.4.

Antes de la fusión, las bases de datos en múltiples particiones divergen, pero se reconcilian en el momento de la fusión.

6.5.2 Fallos y replicación ansiosa

Centrémonos ahora en la replicación ansiosa, que es considerablemente más compleja. Como mencionamos anteriormente, todas las técnicas ansiosas implementan algún tipo de protocolo ROWA, lo que garantiza que, al confirmarse la transacción de actualización, todas las réplicas tengan el mismo valor. La familia de protocolos ROWA es atractiva y elegante. Sin embargo, como vimos al hablar de los protocolos de confirmación, presenta una desventaja importante. Incluso si una de las réplicas no está disponible, la transacción de actualización no puede finalizarse. Por lo tanto, ROWA no cumple uno de los objetivos fundamentales de la replicación: proporcionar una mayor disponibilidad.

Una alternativa a ROWA, que intenta solucionar el problema de baja disponibilidad, es el protocolo ROWA-A (Lectura-Una/Escritura-Todas Disponibles). La idea general es que los comandos de escritura se ejecuten en todas las copias disponibles y la transacción finalice. Las copias que no estaban disponibles en ese momento tendrán que recuperarse cuando lo estén.

Ha habido varias versiones de este protocolo, dos de las cuales comentaremos. El primero se conoce como protocolo de copias disponibles. El coordinador de una transacción de actualización T_i (es decir, el maestro donde se ejecuta la transacción) envía cada $W_i(x)$ a todos los sitios esclavos donde residen réplicas de x y espera la confirmación de la ejecución (o el rechazo). Si se agota el tiempo de espera antes de recibir la confirmación de todos los sitios, considera a los que no han respondido como no disponibles y continúa con la actualización en los sitios disponibles. Los sitios esclavos no disponibles actualizan sus bases de datos al estado más reciente al recuperarse. Sin embargo, cabe destacar que estos sitios podrían no ser conscientes de la existencia de T_i ni de la actualización de x que T_i ha realizado si hubieran dejado de estar disponibles antes del inicio de T_i .

Hay dos complicaciones que deben abordarse. La primera es la posibilidad de que los sitios que el coordinador creía no disponibles estuvieran en funcionamiento y ya hubieran actualizado x , pero su confirmación no le haya llegado al coordinador antes de que se agotara el tiempo. La segunda es que algunos de estos sitios podrían no estar disponibles al inicio de T_i y haberse recuperado desde entonces y haber comenzado a ejecutar transacciones. Por lo tanto, el coordinador realiza un procedimiento de validación antes de confirmar T_i :

El coordinador verifica si todos los sitios que consideraba no disponibles siguen sin estarlo. Para ello, envía un mensaje de consulta a cada uno de ellos. Los que sí lo están responden. Si el coordinador recibe respuesta de uno de estos sitios, cancela T_i , ya que desconoce el estado del sitio previamente no disponible: podría haber estado disponible desde el principio y haber realizado la operación $Wi(x)$, pero su confirmación se retrasó (en cuyo caso todo funciona correctamente), o podría haber estado no disponible al inicio de T_i , pero posteriormente se volvió disponible e incluso haber ejecutado $W_j(x)$ en nombre de otra transacción T_j .

En el último caso, continuar con T_i haría que el programa de ejecución no fuera serializable.

2. Si el coordinador de T no recibe respuesta de ninguno de los sitios que consideraba no disponibles, verifica que todos los sitios disponibles al ejecutar $Wi(x)$ sigan disponibles. De ser así, T puede proceder a la confirmación. Naturalmente, este segundo paso puede integrarse en un protocolo de confirmación.

La segunda variante de ROWA-A que analizaremos es el protocolo ROWA-A distribuido. En este caso, cada sitio S mantiene un conjunto, VS , de sitios que considera disponibles; esta es la "vista" que S tiene de la configuración del sistema. En particular, cuando se envía una transacción T_i , la vista de su coordinador refleja todos los sitios que este sabe que están disponibles (lo denotaremos como $VC(T_i)$ para simplificar).

Se ejecuta una $Ri(x)$ en cualquier réplica en $VC(T_i)$ y una $Wi(x)$ actualiza todas las copias en $VC(T_i)$. El coordinador revisa su vista al final de T_i y, si esta ha cambiado desde su inicio, T_i se cancela. Para modificar V , se ejecuta una transacción atómica especial en todos los sitios, lo que garantiza que no se generen vistas simultáneas. Esto se puede lograr asignando marcas de tiempo a cada V al generarse y garantizando que un sitio solo acepte una nueva vista si su número de versión es mayor que el número de versión de la vista actual de ese sitio.

La clase de protocolos ROWA-A es más resistente a fallas, incluida la partición de la red, que el protocolo ROWA simple.

Otra clase de protocolos de replicación entusiasta son los basados en votación. Las características fundamentales de la votación se presentaron en el capítulo anterior, cuando analizamos la partición de red en bases de datos no replicadas. Las ideas generales se aplican al caso replicado.

Fundamentalmente, cada operación de lectura y escritura debe obtener un número suficiente de votos para poder confirmarse. Estos protocolos pueden ser pesimistas u optimistas. A continuación, analizaremos únicamente los protocolos pesimistas.

Una versión optimista compensa las transacciones para recuperarse si la decisión de confirmación no puede confirmarse al finalizar. Esta versión es adecuada siempre que las transacciones de compensación sean aceptables (véase el capítulo 5).

El algoritmo de votación más antiguo (conocido como algoritmo de Thomas) funciona con bases de datos completamente replicadas y asigna el mismo voto a cada sitio. Para que cualquier operación de una transacción se ejecute, debe obtener los votos afirmativos de la mayoría de los sitios.

Esto se retomó en el algoritmo de Gifford, que también funciona con bases de datos parcialmente replicadas y asigna un voto a cada copia de un elemento de datos replicado. Cada operación debe obtener un quórum de lectura (V_r) o un quórum de escritura (V_w) para leer o escribir un elemento de datos, respectivamente.

Si un elemento de datos tiene un total de V votos, los quórumes deben cumplir las siguientes reglas:

$$1. V_r + V_w > V \quad 2. V_w > V/2$$

Como recordará el lector del capítulo anterior, la primera regla garantiza que un dato no sea leído ni escrito por dos transacciones simultáneamente (evitando así el conflicto de lectura-escritura). La segunda regla, en cambio, garantiza que dos operaciones de escritura de dos transacciones no puedan ocurrir simultáneamente en el mismo dato (evitando así el conflicto de escritura-escritura). Por lo tanto, ambas reglas garantizan la serialización y la equivalencia de una copia.

En el caso de la partición de red, los protocolos basados en quórum funcionan bien, ya que determinan qué transacciones terminarán en función de los votos que obtengan. Las reglas de asignación de votos y umbrales mencionadas anteriormente garantizan que dos transacciones iniciadas en dos particiones diferentes y que accedan a los mismos datos no puedan terminar simultáneamente.

La dificultad de esta versión del protocolo radica en que las transacciones deben obtener quórum incluso para leer datos. Esto ralentiza significativa e innecesariamente el acceso de lectura a la base de datos. A continuación, describimos otro protocolo de votación basado en quórum que soluciona este grave inconveniente de rendimiento.

El protocolo asume ciertas suposiciones sobre la capa de comunicación subyacente y la ocurrencia de fallos. La suposición sobre los fallos es que son "limpios". Esto significa dos cosas:

1. Las fallas que cambian la topología de la red son detectadas por todos los sitios instantáneamente.
2. Cada sitio tiene una vista de la red que consta de todos los sitios con los que puede comunicar.

Basado en la presencia de una red de comunicación que garantiza estas dos condiciones, el protocolo de control de réplicas es una implementación simple del principio ROWA-A. Cuando el protocolo de control de réplicas intenta leer o escribir un dato, primero comprueba si la mayoría de los sitios se encuentran en la misma partición que el sitio donde se ejecuta el protocolo. De ser así, implementa la regla ROWA dentro de esa partición: lee cualquier copia del dato y escribe todas las copias que se encuentren en ella.

Tenga en cuenta que la operación de lectura o escritura se ejecutará solo en una partición. Por lo tanto, este es un protocolo pesimista que garantiza la serialización de una sola copia, pero solo dentro de esa partición. Al reparar la partición, la base de datos se recupera propagando los resultados de la actualización a las demás particiones.

Una pregunta fundamental con respecto a la implementación de este protocolo es si las suposiciones de fallo son realistas. Desafortunadamente, puede que no lo sean, ya que la mayoría de los fallos de red no son "limpios". Existe un retraso entre la ocurrencia de un fallo y su detección por un sitio. Debido a este retraso, es posible que un sitio piense que está en una partición cuando, de hecho, fallos posteriores lo han ubicado en otra. Además, este retraso puede variar según el sitio. Por lo tanto, dos sitios que antes estaban en la misma partición, pero que ahora están en particiones diferentes, pueden continuar durante un tiempo asumiendo que siguen estando en la misma partición. El incumplimiento de estas dos suposiciones de fallo tiene consecuencias negativas significativas en el protocolo de control de réplicas y su capacidad para mantener la serialización de una sola copia.

La solución sugerida consiste en construir, sobre la capa de comunicación física, otra capa de abstracción que oculte las características de fallo "impuro" de la capa de comunicación física y presente al protocolo de control de réplica un servicio de comunicación con propiedades de fallo "limpio". Esta nueva capa de abstracción proporciona particiones virtuales dentro de las cuales opera el protocolo de control de réplica. Una partición virtual es un grupo de sitios que han acordado una visión común de quién está en esa partición. Los sitios se unen y salen de las particiones virtuales bajo el control de esta nueva capa de comunicación, lo que garantiza que se cumplan las suposiciones de fallo limpio.

La ventaja de este protocolo es su simplicidad. No genera sobrecarga para mantener el quórum de accesos de lectura. Por lo tanto, las lecturas pueden avanzar tan rápido como en una red sin particiones. Además, es lo suficientemente general como para que el protocolo de control de réplicas no tenga que diferenciar entre fallos del sitio y particiones de la red.

Dados los métodos alternativos para lograr tolerancia a fallas en el caso de bases de datos replicadas, una pregunta natural es cuáles son las ventajas relativas de estos métodos. Se han realizado varios estudios que analizan estas técnicas, cada uno con diferentes supuestos. Un estudio exhaustivo sugiere que las implementaciones de ROWA-A logran mayor escalabilidad y disponibilidad que las técnicas de quórum.

6.6 Conclusión

En este capítulo, analizamos diferentes enfoques para la replicación de datos y presentamos protocolos adecuados para distintas circunstancias. Cada uno de los protocolos alternativos analizados tiene sus ventajas y desventajas. Los protocolos centralizados Eager son fáciles de implementar, no requieren la coordinación de actualizaciones entre sitios y garantizan la creación de historiales serializables de una sola copia.

Sin embargo, imponen una carga significativa a los sitios maestros, lo que podría convertirlos en cuellos de botella. En consecuencia, son más difíciles de escalar, en particular en la arquitectura de un solo sitio maestro: las versiones de copia primaria tienen mejores propiedades de escalabilidad, ya que las responsabilidades del maestro están en cierta medida distribuidas.

Estos protocolos resultan en tiempos de respuesta largos (los más largos de las cuatro alternativas), ya que

El acceso a cualquier dato debe esperar hasta la confirmación de cualquier transacción que lo esté actualizando (usando 2PC, lo cual es costoso). Además, las copias locales se usan con moderación, solo para operaciones de lectura. Por lo tanto, si la carga de trabajo requiere mucha actualización, es probable que los protocolos centralizados con alta demanda tengan un rendimiento deficiente.

Los protocolos distribuidos con gran entusiasmo también garantizan la serialización de una sola copia y proporcionan una solución simétrica elegante donde cada sitio realiza la misma función. Sin embargo, a menos que el sistema de comunicación admita una multidifusión eficiente, generan un número muy elevado de mensajes que incrementan la carga de la red y los tiempos de respuesta de las transacciones son elevados. Esto también limita su escalabilidad. Además, las implementaciones ingenuas de estos protocolos causarán un número significativo de interbloqueos, ya que las operaciones de actualización se ejecutan en varios sitios simultáneamente.

Los protocolos centralizados perezosos tienen tiempos de respuesta muy cortos, ya que las transacciones se ejecutan y confirman en el servidor maestro, sin necesidad de esperar a que se completen en los servidores esclavos. Además, no es necesario coordinar entre servidores durante la ejecución de una transacción de actualización, lo que reduce el número de mensajes. Por otro lado, no se garantiza la consistencia mutua (es decir, la actualización de los datos en todas las copias), ya que las copias locales pueden estar desactualizadas. Esto significa que no es posible realizar una lectura local con la seguridad de leer la copia más actualizada.

Finalmente, los protocolos multimaestro perezosos ofrecen los tiempos de respuesta más cortos y la mayor disponibilidad. Esto se debe a que cada transacción se ejecuta localmente, sin coordinación distribuida. Solo después de confirmarse, las demás réplicas se actualizan mediante transacciones de actualización. Sin embargo, esta es también la desventaja de estos protocolos: diferentes réplicas pueden actualizarse mediante diferentes transacciones, lo que requiere protocolos de conciliación complejos y provoca la pérdida de actualizaciones.

La replicación se ha estudiado ampliamente tanto en la comunidad de computación distribuida como en la de bases de datos. Si bien existen similitudes considerables en la definición del problema en ambos entornos, también existen diferencias importantes. Quizás las dos diferencias más importantes sean las siguientes.

La replicación de datos se centra en los datos, mientras que la replicación de la computación es igualmente importante en la computación distribuida. En particular, se ha prestado considerable atención a la replicación de datos en entornos móviles que implican operaciones sin conexión. En segundo lugar, la consistencia de las bases de datos y las transacciones es fundamental en la replicación de datos; en la computación distribuida, la consistencia no es una prioridad tan alta. Por consiguiente, se han definido criterios de consistencia considerablemente más débiles.

La replicación se ha estudiado en el contexto de sistemas de bases de datos paralelas, en particular en clústeres de bases de datos paralelas. Se abordan por separado en el capítulo 8.

También nos remitimos al Capítulo 7 sobre cuestiones de replicación que surgen en sistemas de múltiples bases de datos.

6.7 Notas bibliográficas

Los protocolos de replicación y control de réplicas han sido objeto de una investigación significativa desde los inicios de la investigación en bases de datos distribuidas. Este trabajo es...

se resume bien en [Helal et al. 1997]. Los protocolos de control de réplica que tratan la partición de la red se examinan en [Davidson et al. 1985].

Un artículo fundamental que definió un marco para diversos algoritmos de replicación y argumentó que la replicación ansiosa es problemática (lo que abre un torrente de actividad en técnicas perezosas) es [Gray et al. 1996]. La caracterización que utilizamos en este capítulo se basa en este marco. Una caracterización más detallada se presenta en [Wiesmann et al. 2000].

La definición de consistencia eventual proviene de [Saito y Shapiro, 2005], la serialibilidad épsilon se debe a Pu y Leff [1991] y también fue analizada por Ramamritham y Pu [1995] y Wu et al. [1997]. Un estudio reciente sobre técnicas de replicación optimista (o perezosa) se encuentra en [Saito y Shapiro, 2005]. El tema completo se analiza en profundidad en [Kemme et al., 2010].

La frescura, en particular para las técnicas perezosas, ha sido un tema de algunos estudios. En [Pacitti et al.] se analizan técnicas alternativas para garantizar una “mejor” frescura. 1998, 1999, Pacitti y Simon 2000, Röhm et al. 2002, Pape et al. 2004, Akal et al. 2005, Bernstein y otros, 2006].

La extensión del aislamiento de instantáneas a bases de datos replicadas se debe a Lin et al. [2005] y su uso en bases de datos replicadas se analiza en [Plattner y Alonso 2004, Daudjee y Salem 2006]. Bernstein et al. [2006] introducen la serialización RC como otra forma más débil de serialización . La serialización de una sola copia se analiza en [Daudjee y Salem 2004] y el aislamiento de instantáneas en [Daudjee y Salem 2006]. Estos métodos previenen la inversión de transacciones.

Se ha implementado un protocolo de replicación de copia primaria temprana y entusiasta distribuyó INGRES y se describe en [Stonebraker y Neuhold 1977].

En el enfoque de replicación diferida de un solo maestro, el uso de un grafo de replicación para gestionar la ordenación de las transacciones de actualización se debe a Breitbart y Korth [1997]. La gestión de las actualizaciones diferidas mediante la asignación adecuada del sitio primario a los elementos de datos se debe a Chundi et al. [1996].

Bernstein et al. [2006] proponen un algoritmo de replicación perezosa con transmisión completa. paternidad.

El uso de la comunicación grupal se ha discutido en [Chockler et al. 2001, Stanoi et al. 1998, Kemme y Alonso 2000a,b, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. El protocolo distribuido ansioso que discutimos en la Secc. 6.4 se debe a Kemme y Alonso [2000b] y el centralizado perezoso se debe a Pacitti et al. [1999].

El protocolo de copias disponibles en la Sección 6.5.2 se debe a Bernstein y Goodman [1984] y Bernstein y otros [1987].

Existen muchas versiones diferentes de protocolos basados en quórum. Algunas de ellas se analizan en [Triantafillou y Taylor 1995, Paris 1986, Tanenbaum y van Renesse 1988]. El algoritmo de votación inicial fue propuesto por Thomas [1979] y una sugerencia temprana para usar la votación basada en quórum para el control de réplicas se debe a Gifford [1979]. El algoritmo que presentamos en la sección 6.5.2 , que supera los problemas de rendimiento del algoritmo de Gifford, es de El Abbadi et al. [1985]. El estudio exhaustivo que presentamos en la misma sección, que indica los beneficios de ROWA-A, es [Jiménez-Peris et al. 2003]. Además de los algoritmos que hemos descrito aquí, algunos

Otros ejemplos notables se dan en [Davidson 1984, Eager y Sevcik 1983, Herlihy 1987, Minoura y Wiederhold 1982, Skeen y Wright 1984, Wright 1983]. Estos algoritmos generalmente se denominan estáticos , ya que las asignaciones de votos y los quórum de lectura/escritura son fijos a priori. Un análisis de uno de estos protocolos (tales análisis son poco frecuentes) se da en [Kumar y Segev 1993]. Ejemplos de protocolos de replicación dinámica se encuentran en [Jajodia y Mutchler 1987, Barbara et al. 1986, 1989] , entre otros. También es posible cambiar la forma en que se replican los datos. Dichos protocolos se denominan adaptativos y un ejemplo se describe en [Wolfson 1987].

En el presente trabajo se describe un interesante algoritmo de replicación basado en modelos económicos. [Sidell y otros, 1996].

Ceremonias

Problema 6.1 Para cada uno de los cuatro protocolos de replicación (centralizado con avidez, distribuido con avidez, centralizado perezoso, distribuido perezoso), indique un escenario o aplicación donde el enfoque sea más adecuado que los demás. Explique por qué.

Problema 6.2 Una empresa cuenta con varios almacenes distribuidos geográficamente que almacenan y venden productos. Considere el siguiente esquema de base de datos parcial:

ARTÍCULO(ID, NombreArtículo, Precio, ...)

STOCK(ID, Almacén, Cantidad, ...)

CLIENTE(ID, NombreCliente, Dirección, ImporteCrédito, ...)

CLIENTE-PEDIDO(ID, Almacén, Saldo, ...)

PEDIDO(ID, Almacén, IDCliente, Fecha)

LINEA-DE-PEDIDO(ID, ID-artículo, Importe, ...)

La base de datos contiene relaciones con la información del producto (ITEM contiene la información general del producto, STOCK contiene, para cada producto y para cada almacén, el número de piezas actualmente en stock). Además, la base de datos almacena información sobre los clientes, p. ej., la información general sobre los clientes se almacena en la tabla CUSTOMER. Las principales actividades relacionadas con los clientes son el pedido de productos, el pago de facturas y las solicitudes de información general. Existen varias tablas para registrar los pedidos de un cliente. Cada pedido se registra en las tablas ORDER y ORDER-LINE. Para cada pedido/compra, existe una entrada en la tabla de pedidos, que tiene un ID, que indica el ID del cliente, el almacén en el que se envió el pedido, la fecha del pedido, etc. Un cliente puede tener varios pedidos pendientes en un almacén. Dentro de cada pedido, se pueden pedir varios productos. ORDER-LINE contiene una entrada para cada producto del pedido, que puede incluir uno o más productos. CLIENT-ORDER es una tabla resumen que enumera, para cada cliente y para cada almacén, la suma de todos los pedidos existentes.

- (a) La empresa cuenta con un equipo de atención al cliente compuesto por varios empleados que reciben pedidos y pagos, consultan los datos de los clientes locales para emitir facturas o registrar nóminas, etc. Además, responden a cualquier solicitud de los clientes. Por ejemplo, al solicitar productos, se actualizan las tablas PEDIDO-CLIENTE, PEDIDO, LÍNEA DE PEDIDO y STOCK. Para ser flexibles, cada empleado debe poder trabajar con cualquier cliente. Se estima que la carga de trabajo se compone de un 80% de consultas y un 20% de actualizaciones.
- Dado que la carga de trabajo está orientada a consultas, la gerencia decidió construir un clúster de computadoras, cada una equipada con su propia base de datos para agilizar las consultas mediante un acceso local rápido. ¿Cómo replicaría los datos para este propósito? ¿Qué protocolo(s) de control de réplica utilizaría para mantener la consistencia de los datos?
- (b) La gerencia de la empresa debe decidir cada trimestre fiscal sobre su oferta de productos y estrategias de ventas. Para ello, debe observar y analizar continuamente las ventas de los diferentes productos en los distintos almacenes, así como el comportamiento del consumidor. ¿Cómo replicaría los datos para este propósito? ¿Qué protocolo(s) de control de réplica utilizaría para mantener la consistencia de los datos?

Problema 6.3 (*) Una alternativa para garantizar que las transacciones de actualización se puedan aplicar en todos los esclavos en el mismo orden en protocolos de maestro único perezoso con transparencia limitada es el uso de un gráfico de replicación como se analiza en la Sección 6.3.3. Desarrollar un método para la gestión distribuida del gráfico de replicación.

Problema 6.4 Considere los elementos de datos x e y replicados en los sitios de la siguiente manera:

Sitio 1 Sitio 2 Sitio 4 Sitio 3 _____

xx x

y yy

- (a) Asignar votos a cada sitio y otorgar el quórum de lectura y escritura. (b)

Determine las formas posibles en que la red puede particionarse y para cada una especifique en qué grupo de sitios se puede terminar una transacción que actualiza (lee y escribe) x y cuál sería la condición de terminación.

- (c) Repita (b) para y.

Capítulo 7

Integración de bases de datos: múltiples bases de datos

Sistemas



Hasta este punto, consideramos los SGBD distribuidos diseñados de arriba a abajo. En particular, el capítulo 2 se centra en las técnicas de partición y asignación de bases de datos, mientras que el capítulo 4 se centra en el procesamiento distribuido de consultas sobre dichas bases de datos.

Estas técnicas y enfoques son adecuados para SGBD distribuidos, homogéneos y estrechamente integrados. En este capítulo, nos centramos en las bases de datos distribuidas diseñadas de abajo a arriba (en el capítulo 1 las denominamos sistemas multibase de datos). En este caso, ya existen varias bases de datos, y el diseño consiste en integrarlas en una sola. El punto de partida del diseño de abajo a arriba es el conjunto de esquemas conceptuales locales (ECL) individuales. El proceso consiste en integrar las bases de datos locales con sus esquemas (locales) en una base de datos global y generar un esquema conceptual global (ECG) (también denominado esquema mediado).

Consultar en un sistema multibase de datos es más complejo, ya que las aplicaciones y los usuarios pueden realizar consultas mediante el GCS (o las vistas definidas en él) o a través de los LCS, ya que cada base de datos local existente puede tener aplicaciones ejecutándose. Por lo tanto, las técnicas necesarias para el procesamiento de consultas requieren ajustes al enfoque que analizamos en el capítulo 4 , aunque muchas de estas técnicas se mantienen.

La integración de bases de datos y el problema relacionado de las consultas a múltiples bases de datos son solo una parte del problema más general de interoperabilidad , que incluye fuentes de datos no pertenecientes a bases de datos y la interoperabilidad a nivel de aplicación, además del nivel de base de datos. Dividimos este análisis en tres partes: en este capítulo, nos centramos en la integración de bases de datos y los problemas de consulta; en el capítulo 12, abordamos las inquietudes relacionadas con la integración y el acceso a datos web; y en el capítulo 10, bajo el título « Lagos de datos», abordamos el problema más general de la integración de datos de fuentes arbitrarias.

Este capítulo consta de dos secciones principales. En la sección 7.1, analizamos la integración de bases de datos: el proceso de diseño ascendente. En la sección 7.2, analizamos los enfoques para consultar estos sistemas.

7.1 Integración de bases de datos

La integración de bases de datos puede ser física o lógica. En la primera, se integran las bases de datos de origen y la base de datos integrada se materializa. Estos procesos se conocen como almacenes de datos. La integración se realiza mediante herramientas de extracción, transformación y carga (ETL), que permiten la extracción de datos de las fuentes, su transformación para que coincidan con el GCS y su carga (es decir, la materialización). Este proceso se muestra en la figura 7.1.

En la integración lógica, el esquema conceptual global (o mediado) es enteramente virtual y no materializado.

Estos dos enfoques son complementarios y abordan necesidades diferentes. El almacenamiento de datos es compatible con aplicaciones de apoyo a la toma de decisiones, comúnmente denominadas Procesamiento Analítico en Línea (OLAP). Recordemos del capítulo 5 que las aplicaciones OLAP analizan datos históricos resumidos procedentes de diversas bases de datos operativas mediante consultas complejas en tablas potencialmente muy extensas. En consecuencia, los almacenes de datos recopilan datos de diversas bases de datos operativas y los materializan. A medida que se producen actualizaciones en las bases de datos operativas, estas se propagan al almacén de datos, lo que se conoce como mantenimiento de vistas materializadas.

Por el contrario, en la integración lógica de datos, la integración es solo virtual y no existe una base de datos global materializada (véase la Fig. 1.13). Los datos residen en las bases de datos operativas y el GCS proporciona una integración virtual para realizar consultas en las múltiples bases de datos. En estos sistemas, el GCS puede definirse previamente y asignarse a él las bases de datos locales (es decir, los LCS), o puede definirse de abajo a arriba, integrando partes de los LCS de las bases de datos locales. En consecuencia, es posible que el GCS no capture

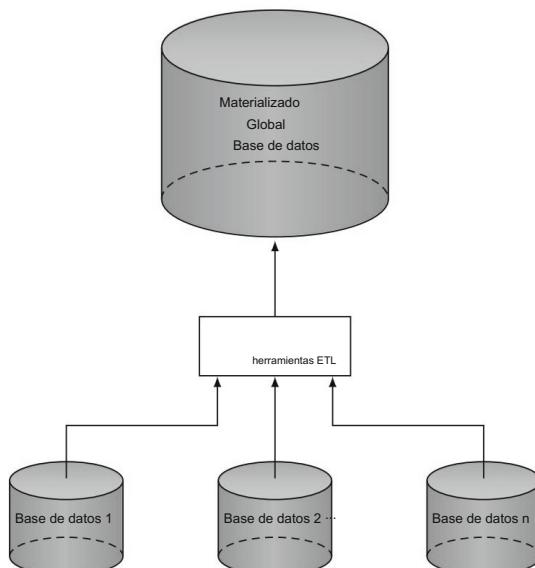


Figura 7.1 Enfoque del almacén de datos

Toda la información en cada uno de los LCS. Las consultas de usuario se formulan sobre este esquema global, que posteriormente se descomponen y se envían a las bases de datos operativas locales para su procesamiento, como se hace en sistemas altamente integrados, con la principal diferencia en la autonomía y la posible heterogeneidad de los sistemas locales. Esto tiene efectos importantes en el procesamiento de consultas, que se analizan en la sección 7.2. Si bien existe un amplio trabajo sobre la gestión de transacciones en estos sistemas, el soporte de actualizaciones globales resulta bastante difícil dada la autonomía de los SGBD operativos subyacentes.

Por lo tanto, son principalmente de sólo lectura.

La integración lógica de datos y los sistemas resultantes se conocen con diversos nombres; integración de datos e integración de información son quizás los términos más comunes en la literatura, aunque generalmente se refieren a más que la integración de bases de datos e incorporan datos de diversas fuentes. En este capítulo, nos centramos en la integración de bases de datos autónomas y (posiblemente) heterogéneas; por lo tanto, utilizaremos el término integración de bases de datos o sistemas multibase de datos (MDBS).

7.1.1 Metodología de diseño de abajo hacia arriba

El diseño de abajo hacia arriba implica el proceso mediante el cual los datos de las bases de datos participantes pueden integrarse (física o lógicamente) para formar una única base de datos global cohesiva. Como se mencionó anteriormente, en algunos casos, se define primero el esquema conceptual global (o mediado), en cuyo caso el diseño ascendente implica la asignación de los LCS a este esquema. En otros casos, el GCS se define como una integración de partes de los LCS. En este caso, el diseño ascendente implica tanto la generación del GCS como la asignación de los LCS individuales a este.

Si el GCS se define previamente, la relación entre el GCS y los LCS puede ser de dos tipos fundamentales: local como vista y global como vista. En los sistemas local como vista (LAV), la definición del GCS existe y cada LCS se trata como una definición de vista sobre él. En los sistemas global como vista (GAV), por otro lado, el GCS se define como un conjunto de vistas sobre los LCS. Estas vistas indican cómo se pueden derivar los elementos del GCS, cuando sea necesario, a partir de los elementos de los LCS. Una forma de entender la diferencia entre ambos es en términos de los resultados que se pueden obtener de cada sistema.

En GAV, los resultados de las consultas se limitan al conjunto de objetos definidos en el GCS, aunque los SGBD locales pueden ser considerablemente más completos (Fig. 7.2a). En LAV, por otro lado, los resultados se limitan a los objetos de los SGBD locales, mientras que la definición del GCS puede ser más completa (Fig. 7.2b). Por lo tanto, en los sistemas LAV, puede ser necesario gestionar respuestas incompletas. También se ha propuesto una combinación de estos dos enfoques, denominada global-local-como-vista (GLAV), donde la relación entre el GCS y los LCS se especifica utilizando tanto LAV como GAV.

El diseño ascendente se realiza en dos pasos generales (Fig. 7.3): traducción del esquema (o simplemente traducción) y generación del esquema. En el primer paso, los esquemas de la base de datos de componentes se traducen a una representación canónica intermedia común ($InS1, InS2, \dots, InSn$). El uso de una representación canónica facilita la traducción.

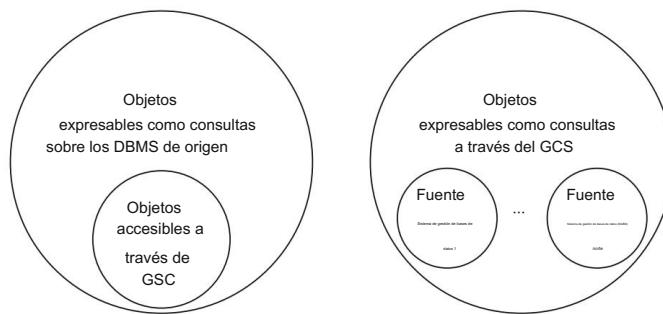


Fig. 7.2 Mapeos de GAV y LAV (basados en [Koch 2001])

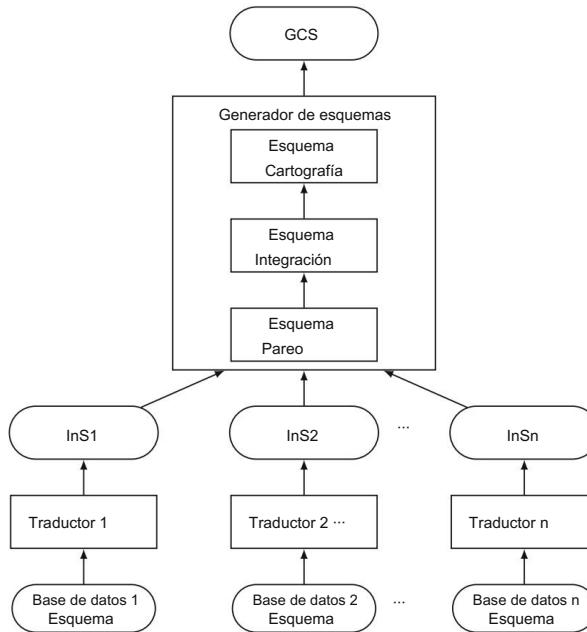


Fig. 7.3 Proceso de integración de bases de datos

El proceso de lació n se simplifica al reducir el número de traductores necesarios. La elección del modelo canónico es importante. En principio, debe ser lo suficientemente expresivo como para incorporar los conceptos disponibles en todas las bases de datos que se integrarán posteriormente. Entre las alternativas utilizadas se incluyen el modelo entidad-relación, el modelo orientado a objetos o un grafo que puede simplificarse a un trie o XML. En este capítulo, utilizaremos simplemente el modelo relacional como nuestro modelo de datos canónico, a pesar de sus conocidas deficiencias para representar conceptos semánticos complejos. Esta elección no afecta de ninguna manera fundamental la discusión de los temas principales.

Problemas de integración de datos. En cualquier caso, no abordaremos los detalles de la traducción de diversos modelos de datos a relacionales; esto se puede encontrar en muchos libros de texto sobre bases de datos.

Claramente, el paso de traducción solo es necesario si las bases de datos componentes son heterogéneas y los esquemas locales se definen utilizando diferentes modelos de datos. Se ha trabajado en el desarrollo de la federación de sistemas, en la que sistemas con modelos de datos similares se integran (p. ej., los sistemas relacionales se integran en un esquema conceptual y, quizás, las bases de datos de objetos se integran en otro esquema) y estos esquemas integrados se combinan posteriormente (p. ej., el proyecto AURORA). En este caso, el paso de traducción se retrasa, lo que proporciona mayor flexibilidad a las aplicaciones para acceder a las fuentes de datos subyacentes de forma adecuada a sus necesidades.

En el segundo paso del diseño de abajo hacia arriba, se utilizan los esquemas intermedios para Generar un GCS. El proceso de generación de esquemas consta de los siguientes pasos:

1. Coincidencia de esquemas para determinar las correspondencias sintácticas y semánticas entre los elementos LCS traducidos o entre elementos LCS individuales y los elementos GCS predefinidos (Sección 7.1.2).
2. Integración de los elementos del esquema común en un esquema conceptual global (mediado) si aún no se ha definido uno (Sección 7.1.3).
3. Mapeo de esquemas que determina cómo mapear los elementos de cada LCS a los otros elementos del GCS (Sección 7.1.4).

También es posible dividir el paso de mapeo de esquemas en dos fases: generación de restricciones de mapeo y generación de transformaciones. En la primera fase, dadas las correspondencias entre dos esquemas, se genera una función de transformación, como una consulta o una definición de vista sobre el esquema de origen, que llenaría el esquema de destino. En la segunda fase, se genera un código ejecutable correspondiente a esta función de transformación, que generaría una base de datos de destino coherente con estas restricciones. En algunos casos, las restricciones se incluyen implícitamente en las correspondencias, eliminando así la necesidad de la primera fase.

Ejemplo 7.1 Para facilitar nuestra discusión sobre el diseño de esquemas globales en sistemas multibase de datos, utilizaremos un ejemplo que es una extensión de la base de datos de ingeniería que hemos utilizado a lo largo del libro. Para demostrar ambas fases del proceso de integración de bases de datos, introducimos cierta heterogeneidad del modelo de datos en nuestro ejemplo.

Consideremos dos organizaciones, cada una con sus propias definiciones de base de datos. Una es el ejemplo de base de datos relacional que presentamos en el capítulo 2. Repetimos esa definición en la figura 7.4 para mayor claridad. La segunda base de datos también define datos similares, pero se especifica según el modelo de datos entidad-relación (ER) como representado en la figura 7.5.¹

Suponemos que el lector está familiarizado con el modelo de datos entidad-relación. Por lo tanto, no describiremos el formalismo, excepto para hacer las siguientes observaciones sobre la semántica de la figura 7.5. Esta base de datos es similar a la relacional.

¹En este capítulo, continuamos nuestra notación de nombres de relaciones de composición tipográfica en fuente de máquina de escribir, pero utilizaremos una fuente normal para los componentes del modelo ER para poder diferenciarlos fácilmente.

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Fig. 7.4 Representación de la base de datos de ingeniería relacional

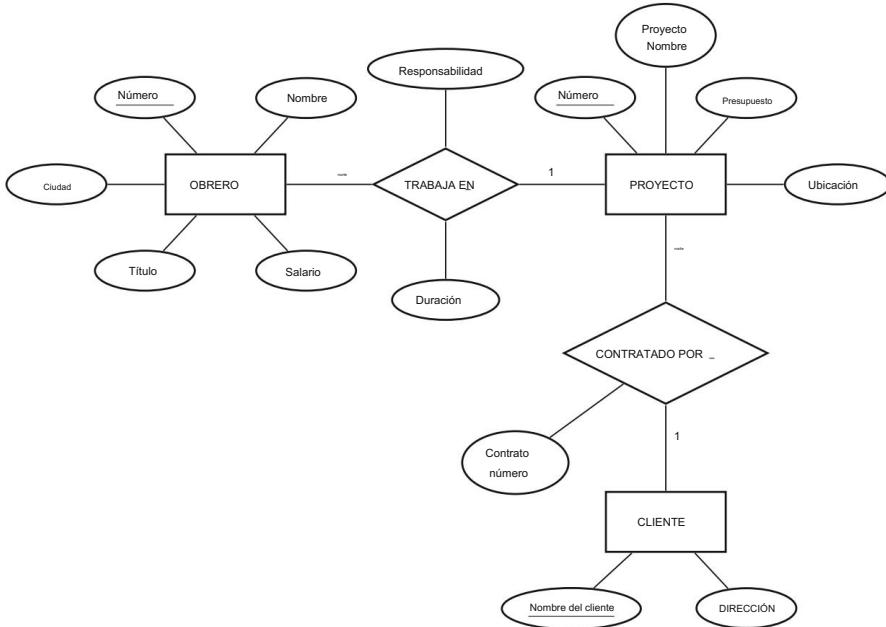


Fig. 7.5 Base de datos entidad-relación

Definición de la base de datos de ingeniería de la Fig. 7.4, con una diferencia significativa: También mantiene datos sobre los clientes para quienes se realizan los proyectos. Los cuadros rectangulares en la Fig. 7.5 representan las entidades modeladas en la base de datos, y los diamantes indican una relación entre las entidades a las que están asociados. Conectado. El tipo de relación se indica alrededor de los rombos. Por ejemplo, La relación CONTRATADO POR es de muchos a uno desde la entidad PROYECTO a la Entidad CLIENTE (por ejemplo, cada proyecto tiene un solo cliente, pero cada cliente puede tener muchos proyectos). De manera similar, la relación WORKS-IN indica una relación de muchos a muchos. Relación entre las dos relaciones conectadas. Los atributos de las entidades y la Las relaciones se muestran como elipses.

Ejemplo 7.2 La correspondencia del modelo ER con el modelo relacional se da en Fig. 7.6. Nótese que hemos renombrado algunos atributos para asegurar el nombre. unicidad.

```

WORKER(WNUMBER, NAME, TITLE, SALARY,CITY)
PROJECT(PNUMBER, PNAME, BUDGET)
CLIENT(CNAME, ADDRESS)
WORKS_IN(WNUMBER, PNUMBER, RESPONSIBILITY, DURATION)
CONTRACTED_BY(PNUMBER, CNAME, CONTRACTNO)

```

Fig. 7.6 Mapeo relacional del esquema ER

7.1.2 Coincidencia de esquemas

Dados dos esquemas, la coincidencia de esquemas determina, para cada concepto de uno, qué concepto del otro coincide con él. Como se mencionó anteriormente, si el GCS ya se ha definido, uno de estos esquemas suele ser el GCS, y la tarea consiste en hacer coincidir cada LCS con el GCS. De lo contrario, la coincidencia se realiza sobre dos LCS. Las coincidencias determinadas en esta fase se utilizan en la asignación de esquemas para generar un conjunto de asignaciones dirigidas que, al aplicarse al esquema de origen, asignarían sus conceptos al esquema de destino.

Las coincidencias que se definen o descubren durante la coincidencia de esquemas se especifican como un conjunto de reglas donde cada regla (r) identifica una correspondencia (c) entre dos elementos, un predicado (p) que indica cuándo puede mantenerse la correspondencia y un valor de similitud (s) entre los dos elementos identificados en la correspondencia. Una correspondencia puede simplemente identificar que dos conceptos son similares (lo que denotaremos por \approx) o puede ser una función que especifica que un concepto puede derivarse por un cálculo sobre el otro (por ejemplo, si el valor del presupuesto de un proyecto se especifica en dólares estadounidenses, mientras que el otro se especifica en euros, la correspondencia puede especificar que uno se obtiene multiplicando el otro por el tipo de cambio apropiado). El predicado es una condición que califica la correspondencia al especificar cuándo podría cumplirse. Por ejemplo, en el ejemplo de presupuesto especificado anteriormente, p puede especificar que la regla se cumple solo si la ubicación de un proyecto está en EE. UU., mientras que el otro está en la zona euro. El valor de similitud para cada regla se puede especificar o calcular. Los valores de similitud son valores reales en el rango $[0,1]$. Por lo tanto, un conjunto de coincidencias se puede definir como $M = \{r\}$, donde $r = c, p, s$.

Como se indicó anteriormente, las correspondencias pueden descubrirse o especificarse. Si bien es deseable automatizar este proceso, existen numerosos factores que lo complican. El más importante es la heterogeneidad del esquema, que se refiere a las diferencias en la forma en que se capturan los fenómenos del mundo real en distintos esquemas. Este es un tema crucial, al que dedicamos una sección aparte (Sección 7.1.2.1). Además de la heterogeneidad del esquema, otros problemas que complican el proceso de emparejamiento son los siguientes:

- Información insuficiente del esquema y de la instancia: Los algoritmos de coincidencia dependen de la información que se puede extraer del esquema y de las instancias de datos existentes. En algunos casos, puede haber ambigüedad debido a la información insuficiente.

Información proporcionada sobre estos elementos. Por ejemplo, el uso de nombres cortos o abreviaturas ambiguas para conceptos, como hemos hecho en nuestros ejemplos, puede dar lugar a coincidencias incorrectas.

- Falta de documentación del esquema: En la mayoría de los casos, los esquemas de la base de datos no están bien documentados o no están documentados en absoluto. Con frecuencia, el diseñador del esquema ya no está disponible para guiar el proceso. La falta de estas fuentes de información vitales dificulta la correspondencia.

Subjetividad de la correspondencia: Finalmente, es importante reconocer que la correspondencia de elementos del esquema puede ser muy subjetiva; dos diseñadores pueden no coincidir en una única asignación "correcta". Esto dificulta considerablemente la evaluación de la precisión de un algoritmo dado.

Sin embargo, se han desarrollado enfoques algorítmicos para el problema de emparejamiento, que abordamos en esta sección. Diversos aspectos afectan al algoritmo de emparejamiento en particular. Los más importantes son los siguientes:

- Coincidencia de esquemas e instancias. Hasta ahora, en este capítulo, nos hemos centrado en la integración de esquemas; por lo tanto, nuestra atención se ha centrado, naturalmente, en la coincidencia de conceptos de un esquema con los de otro. Se han desarrollado numerosos algoritmos que funcionan con elementos de esquema. Sin embargo, existen otros que se han centrado en las instancias de datos o en una combinación de información de esquema e instancias de datos. El argumento es que considerar las instancias de datos puede ayudar a resolver algunos de los problemas semánticos mencionados anteriormente. Por ejemplo, si el nombre de un atributo es ambiguo, como en "contact-info", obtener sus datos puede ayudar a identificar su significado; si sus instancias de datos tienen el formato de número de teléfono, obviamente es el número de teléfono del agente de contacto, mientras que las cadenas largas pueden indicar que es el nombre del agente de contacto. Además, existe una gran cantidad de atributos, como códigos postales, nombres de países y direcciones de correo electrónico, que se pueden definir fácilmente a través de sus instancias de datos.

La coincidencia que se basa únicamente en la información del esquema puede ser más eficiente, porque no requiere una búsqueda en instancias de datos para hacer coincidir los atributos. Además, este enfoque es el único viable cuando hay pocas instancias de datos disponibles en las bases de datos coincidentes, en cuyo caso el aprendizaje puede no ser confiable.

Sin embargo, en algunos casos, por ejemplo, los sistemas peer-to-peer (véase Cap. 9), puede que no exista un esquema, en cuyo caso la correspondencia basada en instancias es el único enfoque apropiado.

- Nivel de elemento vs. nivel de estructura. Algunos algoritmos de coincidencia operan sobre elementos individuales del esquema, mientras que otros también consideran las relaciones estructurales entre estos elementos. El concepto básico del enfoque a nivel de elemento es que la mayor parte de la semántica del esquema se captura mediante los nombres de los elementos. Sin embargo, esto puede fallar al encontrar asignaciones complejas que abarquen múltiples atributos. Los algoritmos de coincidencia que también consideran la estructura se basan en la creencia de que, normalmente, las estructuras de los esquemas coincidentes tienden a ser similares.
- Cardinalidad de coincidencia. Los algoritmos de coincidencia presentan diversas capacidades en cuanto a la cardinalidad de las asignaciones. Los enfoques más sencillos utilizan la asignación 1:1, lo que significa que cada elemento de un esquema se corresponde con exactamente un elemento de...

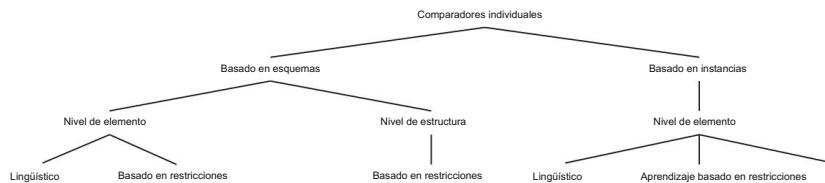


Figura 7.7 Taxonomía de las técnicas de coincidencia de esquemas

El otro esquema. La mayoría de los algoritmos propuestos pertenecen a esta categoría, ya que los problemas se simplifican considerablemente. Claro que hay muchos casos en los que esta suposición no es válida. Por ejemplo, un atributo llamado "Precio total" podría asignarse a la suma de dos atributos en otro esquema llamados "Subtotal" e "Impuestos". Estas asignaciones requieren algoritmos de coincidencia más complejos que consideren asignaciones 1:M y N:M.

Estos criterios, entre otros, pueden utilizarse para elaborar una taxonomía de enfoques de emparejamiento. Según esta taxonomía (que seguiremos en este capítulo con algunas modificaciones), el primer nivel de separación se establece entre los comparadores basados en esquemas y los basados en instancias (Fig. 7.7). Los comparadores basados en esquemas pueden clasificarse a su vez en niveles de elemento y de estructura, mientras que para los enfoques basados en instancias, solo las técnicas a nivel de elemento son relevantes. En el nivel más bajo, las técnicas se caracterizan como lingüísticas o basadas en restricciones. Es en este nivel donde se presentan las diferencias fundamentales entre los algoritmos de emparejamiento, y nos centraremos en ellos en el resto del capítulo, analizando los enfoques lingüísticos en la sección 7.1.2.2, los basados en restricciones en la sección 7.1.2.3 y las técnicas basadas en aprendizaje en la sección 7.1.2.4. Estos se denominan enfoques de emparejamiento individual, y sus combinaciones son posibles mediante el desarrollo de comparadores híbridos o compuestos (Sección 7.1.2.5).

7.1.2.1 Heterogeneidad del esquema

Los algoritmos de coincidencia de esquemas abordan tanto la heterogeneidad estructural como la semántica entre los esquemas emparejados. Las analizamos en esta sección antes de presentar los diferentes algoritmos de coincidencia.

Los conflictos estructurales se producen de cuatro maneras posibles: como conflictos de tipo, conflictos de dependencia, conflictos de clave o conflictos de comportamiento. Los conflictos de tipo ocurren cuando el mismo objeto se representa mediante un atributo en un esquema y una entidad (relación) en otro. Los conflictos de dependencia ocurren cuando se utilizan diferentes modos de relación (p. ej., uno a uno frente a muchos a muchos) para representar lo mismo en diferentes esquemas. Los conflictos de clave ocurren cuando hay diferentes claves candidatas disponibles y se seleccionan diferentes claves primarias en diferentes esquemas. Los conflictos de comportamiento están implícitos en el mecanismo de modelado. Por ejemplo, eliminar el último elemento de un

La eliminación de la base de datos puede provocar la eliminación de la entidad que la contiene (es decir, la eliminación del último empleado provoca la disolución del departamento).

Ejemplo 7.3 En el ejemplo de ejecución de este capítulo, tenemos dos conflictos estructurales. El primero es un conflicto de tipos que involucra a clientes de proyectos. En el esquema de la Fig. 7.5, el cliente de un proyecto se modela como una entidad. Sin embargo, en el esquema de la Fig. 7.4, el cliente se incluye como un atributo de la entidad PROJ.

El segundo conflicto estructural es un conflicto de dependencia que involucra la relación WORKS_IN (Fig. 7.5) y la relación ASG (Fig. 7.4). En la primera, la relación es de muchos a uno desde el TRABAJADOR hasta el PROYECTO, mientras que en la segunda, la relación es de muchos a muchos.

Las diferencias estructurales entre esquemas son importantes, pero su identificación y resolución no son suficientes. La comparación de esquemas debe considerar la semántica (posiblemente diferente) de los conceptos del esquema. Esto se conoce como heterogeneidad semántica, un término confuso y sin una definición clara. Básicamente, se refiere a las diferencias entre las bases de datos relacionadas con el significado, la interpretación y el uso previsto de los datos. Existen intentos de formalizar la heterogeneidad semántica y establecer su vínculo con la heterogeneidad estructural; adoptaremos un enfoque más informal y analizaremos algunos de los problemas de la heterogeneidad semántica de forma intuitiva. A continuación, se presentan algunos de estos problemas que los algoritmos de comparación deben abordar.

- Sinónimos, homónimos, hiperónimos. Los sinónimos son términos múltiples que se refieren al mismo concepto. En nuestro ejemplo de base de datos, la relación PROJ y la entidad PROJECT se refieren al mismo concepto. Los homónimos, por otro lado, ocurren cuando el mismo término se usa para significar cosas diferentes en diferentes contextos. De nuevo, en nuestro ejemplo, PRESUPUESTO puede referirse al presupuesto bruto en una base de datos y al presupuesto neto (después de alguna deducción de gastos generales) en otra, lo que dificulta su simple comparación. Hiperónimo es un término que es más genérico que una palabra similar. Aunque no hay un ejemplo directo de ello en las bases de datos que estamos considerando, el concepto de un vehículo en una base de datos es un hiperónimo para el concepto de un automóvil en otra (incidentalmente, en este caso, Auto es un hipónimo de Vehículo). Estos problemas pueden abordarse mediante el uso de ontologías de dominio que definen la organización de conceptos y términos en un dominio particular.
 - Ontologías diferentes: Incluso si se utilizan ontologías de dominio para abordar problemas en un dominio, es frecuente que sea necesario cotejar esquemas de diferentes dominios. En este caso, es necesario tener cuidado con el significado de los términos en las distintas ontologías, ya que pueden depender en gran medida del dominio. Por ejemplo, un atributo llamado CARGA puede implicar una medida de resistencia en una ontología eléctrica, pero en una ontología mecánica, puede representar una medida de peso.
- Redacción imprecisa: Los esquemas pueden contener nombres ambiguos. Por ejemplo, los atributos LOCATION (de ER) y LOC (de relacional) en nuestra base de datos de ejemplo pueden referirse a la dirección completa o solo a una parte de ella. De igual manera, un atributo llamado "contact-info" puede implicar que contiene el nombre del agente de contacto o su número de teléfono. Este tipo de ambigüedades son comunes.

7.1.2.2 Enfoques de correspondencia lingüística

Los métodos de correspondencia lingüística, como su nombre indica, utilizan nombres de elementos y otra información textual (como descripciones o anotaciones textuales en las definiciones de esquemas) para realizar correspondencias entre elementos. En muchos casos, pueden utilizar fuentes externas, como tesauros, para facilitar el proceso.

Las técnicas lingüísticas pueden aplicarse tanto en enfoques basados en esquemas como en instancias. En el primer caso, las similitudes se establecen entre los elementos del esquema, mientras que en el segundo, se especifican entre los elementos de instancias de datos individuales. Para centrar nuestra discusión, consideraremos principalmente los enfoques de comparación lingüística basados en esquemas, mencionando brevemente las técnicas basadas en instancias. Por consiguiente, utilizaremos la notación $SC1.\text{elemento-1} \approx SC2.\text{elemento-2}, p, s$ para representar que el elemento-1 del esquema SC1 corresponde al elemento-2 del esquema SC2 si se cumple el predicado p , con un valor de similitud de s . Los comparadores utilizan estas reglas y valores de similitud para determinar el valor de similitud de los elementos del esquema.

Los comparadores lingüísticos que operan a nivel de elemento del esquema suelen procesar los nombres de dichos elementos y gestionar casos como sinónimos, homónimos e hiperónimos. En algunos casos, las definiciones del esquema pueden incluir anotaciones (comentarios en lenguaje natural) que los comparadores lingüísticos pueden aprovechar. En el caso de los enfoques basados en instancias, los comparadores lingüísticos se centran en técnicas de recuperación de información como frecuencias de palabras, términos clave, etc. En estos casos, los comparadores "deducen" similitudes basándose en estas medidas de recuperación de información.

Los comparadores lingüísticos de esquemas utilizan un conjunto de reglas lingüísticas (también llamadas terminológicas) que pueden crearse manualmente o descubrirse mediante fuentes de datos auxiliares como tesauros, por ejemplo, WordNet. En el caso de las reglas creadas manualmente, el diseñador también debe especificar el predicado p y el valor de similitud s . En el caso de las reglas descubiertas, estas pueden ser especificadas por un experto tras el descubrimiento o pueden calcularse mediante alguna de las técnicas que analizaremos en breve.

Las reglas lingüísticas personalizadas pueden abordar cuestiones como el uso de mayúsculas, las abreviaturas y las relaciones conceptuales. En algunos sistemas, el diseñador especifica estas reglas individualmente para cada esquema (reglas intraesquemáticas), y el algoritmo de coincidencia descubre las reglas interesquemáticas. Sin embargo, en la mayoría de los casos, la base de reglas contiene reglas intraesquemáticas e interesquemáticas.

Ejemplo 7.4 En la base de datos relacional del Ejemplo 7.2, el conjunto de reglas puede haberse definido (de manera bastante intuitiva) de la siguiente manera, donde RelDB se refiere al esquema relacional y ERDB se refiere al esquema ER traducido:

```
    nombres en mayúsculas ≈ nombres en minúsculas, verdadero, 1.0)
    nombres en mayúsculas ≈ nombres en mayúsculas, verdadero, 1.0)
    nombres en mayúsculas ≈ nombres en minúsculas, verdadero, 1.0)
    RelDB.ASG ≈ ERDB.WORKS_IN, verdadero, 0.8
    ...

```

Las tres primeras reglas son genéricas y especifican cómo gestionar las mayúsculas, mientras que la cuarta especifica una similitud entre el ASG de RelDB y el WORKS_IN de ERDB. Dado que estas correspondencias siempre se cumplen, p = verdadero.

Como se indicó anteriormente, existen maneras de determinar automáticamente la similitud de los nombres de los elementos. Por ejemplo, COMA utiliza las siguientes técnicas para determinar la similitud de dos nombres de elementos:

- Los afijos que son los prefijos y sufijos comunes entre los dos elementos.

Se determinan las cadenas de nombres.

Se comparan los n-gramas de las dos cadenas de nombres de elementos. Un n-grama es una subcadena de longitud n y la similitud es mayor si las dos cadenas tienen más n-gramas en común.

Se calcula la distancia de edición entre dos cadenas de nombres de elementos. Esta distancia (también llamada métrica de Levenshtein) determina el número de modificaciones de caracteres (adiciones, eliminaciones e inserciones) que se deben realizar en una cadena para convertirla en la segunda.

Se calcula el código soundex de los nombres

de los elementos. Esto proporciona la similitud fonética entre los nombres basándose en sus códigos soundex. El código soundex de las palabras en inglés se obtiene mediante la conversión de la palabra en una letra y tres números. Este valor hash corresponde (aproximadamente) a cómo sonaría la palabra. Lo importante de este código en nuestro contexto es que dos palabras con un sonido similar tendrán códigos soundex similares.

Ejemplo 7.5 Considere la coincidencia de los atributos RESP y RESPONSABILIDAD en los dos esquemas de ejemplo que estamos considerando. Las reglas definidas en el Ejemplo 7.4 solucionan las diferencias de mayúsculas y minúsculas, por lo que solo nos queda la coincidencia de RESP con RESPONSABILIDAD. Consideremos cómo se puede calcular la similitud entre las dos cadenas utilizando los métodos de distancia de edición y n-gramas.

La cantidad de cambios de edición que uno necesita hacer para convertir una de estas cadenas en otra es 10 (ya sea agregamos los caracteres "O", "N", "S", "I", "B", "I", "L", "I", "T", "Y" a la cadena "RESP" o eliminar los mismos caracteres de la cadena "RESPONSABILIDAD"). Por lo tanto, la proporción de los cambios requeridos es 10/14, lo que define la distancia de edición entre estas dos cadenas; $1 - (10/14) = 4/14 = 0,29$ es su similitud.

Para el cálculo de n-gramas, primero debemos determinar el valor de n. En este ejemplo, supongamos que n = 3, por lo que buscamos 3-gramas. Los 3-gramas de la cadena "RESP" son "RES" y "ESP". De forma similar, hay doce 3-gramas de "RESPONSABILIDAD": "RES", "ESP", "SPO", "PON", "ONS", "NSI", "SIB", "IBI", "BIP", "ILI", "LIT" e "ITY". De doce, hay dos 3-gramas coincidentes, lo que da una similitud de $2/12 = 0,17$.

Los ejemplos que hemos cubierto en esta sección se clasifican en la categoría de coincidencias 1:1: se ha emparejado un elemento de un esquema específico con un elemento de otro esquema. Como se mencionó anteriormente, es posible tener coincidencias 1:N (p. ej., los valores de los elementos Dirección, Ciudad y País de una base de datos se pueden extraer de un solo elemento Dirección de otra), N:1 (p. ej., el precio total se puede calcular a partir del subtotal y los impuestos).

elementos) o N:M (por ejemplo, Book_title, la información de calificación se puede extraer mediante una unión de dos tablas, una de las cuales contiene información del libro y la otra mantiene las revisiones y calificaciones de los lectores). Los comparadores 1:1, 1:N y N:1 se utilizan normalmente en la coincidencia a nivel de elemento, mientras que la coincidencia a nivel de esquema también puede utilizar la coincidencia N:M, ya que, en el último caso, la información de esquema necesaria está disponible.

7.1.2.3 Enfoques de emparejamiento basados en restricciones

Las definiciones de esquema casi siempre contienen información semántica que restringe los valores en la base de datos. Esta información suele incluir información sobre el tipo de datos, rangos permitidos para los valores de datos, restricciones de clave, etc. En el caso de las técnicas basadas en instancias, se pueden extraer los rangos de valores existentes, así como algunos patrones presentes en los datos de instancia. Estos pueden ser utilizados por los comparadores.

Consideré tipos de datos que capturan una gran cantidad de información semántica. Esta información puede utilizarse para desambiguar conceptos y también para enfocar la coincidencia.

Por ejemplo, RESP y RESPONSABILIDAD presentan valores de similitud relativamente bajos, según los cálculos del Ejemplo 7.5. Sin embargo, si comparten la misma definición de tipo de dato, esto puede aumentar su valor de similitud. De igual forma, la comparación de tipos de dato puede diferenciar entre elementos con alta similitud léxica. Por ejemplo, ENO en la Fig. 7.4 tiene la misma distancia de edición y valores de similitud de n-gramas que los dos atributos NÚMERO de la Fig. 7.5 (por supuesto, nos referimos a los nombres de estos atributos). En este caso, los tipos de dato pueden ser útiles: si el tipo de dato de ENO y el número de trabajador (WORKER.NUMBER) es entero, mientras que el tipo de dato de número de proyecto (PROJECT.NUMBER) es una cadena, la probabilidad de que ENO coincida con WORKER.NUMBER es significativamente mayor.

En los enfoques basados en la estructura, las similitudes estructurales entre los dos esquemas pueden aprovecharse para determinar la similitud de sus elementos. Si dos elementos son estructuralmente similares, aumenta nuestra confianza en que representan el mismo concepto. Por ejemplo, si dos elementos tienen nombres muy diferentes y no hemos podido establecer su similitud mediante comparadores de elementos, pero comparten las mismas propiedades (p. ej., los mismos atributos) y los mismos tipos de datos, podemos tener mayor confianza en que estos dos elementos podrían representar el mismo concepto.

La determinación de la similitud estructural implica comprobar la similitud de las "vecindades" de los dos conceptos en consideración. La definición de la vecindad se realiza típicamente mediante una representación gráfica de los esquemas donde cada concepto (relación, entidad, atributo) es un vértice y existe una arista dirigida entre dos vértices si y solo si los dos conceptos están relacionados (por ejemplo, existe una arista desde un vértice de relación a cada uno de sus atributos, o existe una arista desde un vértice de atributo de clave foránea al vértice de atributo de clave primaria al que hace referencia). En este caso, la vecindad se puede definir en términos de los vértices que se pueden alcanzar dentro de una cierta longitud de camino de cada concepto, y el problema se reduce a comprobar la similitud de los subgrafos en esta vecindad. Muchos de estos algoritmos consideran el trie con raíz en el concepto que se está examinando y calculan el

Similitud de los conceptos representados por los vértices raíz en los dos árboles. La idea fundamental es que si los subgrafos (subárboles) son similares, esto aumenta la similitud de los conceptos representados por el vértice "raíz" en los dos grafos. La similitud de los subgrafos se determina típicamente en un proceso ascendente, comenzando en las hojas cuya similitud se determina mediante la coincidencia de elementos (por ejemplo, similitud de nombres a nivel de sinónimos o compatibilidad de tipos de datos). La similitud de los dos subárboles se determina recursivamente con base en la similitud de los vértices en el subárbol. La similitud de dos subgrafos (subárboles) se define entonces como la fracción de hojas en los dos subárboles que están fuertemente vinculadas. Esto se basa en el supuesto de que los vértices de las hojas contienen más información y que la similitud estructural de dos elementos de esquema no hojas está determinada por la similitud de los vértices de las hojas en sus respectivos subárboles, incluso si sus hijos inmediatos no son similares. Estas son reglas heurísticas y es posible definir otras.

Otro enfoque interesante para considerar la vecindad en grafos dirigidos al calcular la similitud de vértices es la inundación de similitud. Esta técnica parte de un grafo inicial donde las similitudes de vértices ya están determinadas mediante un comparador de elementos y se propaga iterativamente para determinar la similitud de cada vértice con sus vecinos. Por lo tanto, siempre que dos elementos en dos esquemas resulten similares, la similitud de sus vértices adyacentes aumenta. El proceso iterativo se detiene cuando las similitudes de vértices se estabilizan. En cada iteración, para reducir la carga de trabajo, se selecciona un subconjunto de vértices como las coincidencias más plausibles, que se consideran en la iteración posterior.

Ambos enfoques son independientes de la semántica de las aristas. En algunas representaciones gráficas, estas aristas tienen semántica adicional. Por ejemplo, las aristas de contención desde el vértice de una relación o entidad hasta sus atributos pueden distinguirse de las aristas referenciales desde el vértice de un atributo de clave foránea hasta el vértice de un atributo de clave primaria correspondiente. Algunos sistemas (p. ej., DIKE) aprovechan esta semántica de aristas.

7.1.2.4 Emparejamiento basado en el aprendizaje

Un tercer enfoque alternativo propuesto consiste en utilizar técnicas de aprendizaje automático para determinar la coincidencia de esquemas. Los enfoques basados en el aprendizaje plantean el problema como uno de clasificación, donde los conceptos de diversos esquemas se clasifican en clases según su similitud. Esta similitud se determina comprobando las características de las instancias de datos de las bases de datos que corresponden a estos esquemas.

La clasificación de conceptos según sus características se aprende estudiando las instancias de datos en un conjunto de datos de entrenamiento.

El proceso es el siguiente (Fig. 7.8). Se prepara un conjunto de entrenamiento (T) que consta de instancias de correspondencias de ejemplo entre los conceptos de dos bases de datos D_i y D_j .

Este conjunto de entrenamiento se puede generar tras la identificación manual de las correspondencias de esquema entre dos bases de datos, seguida de la extracción de instancias de datos de entrenamiento de ejemplo o mediante la especificación de una expresión de consulta que convierte los datos de una base de datos a otra. El alumno utiliza estos datos de entrenamiento para adquirir datos probabilísticos.

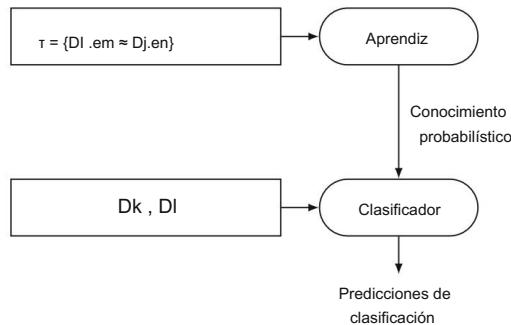


Figura 7.8 Enfoque de emparejamiento basado en el aprendizaje

Información sobre las características de los conjuntos de datos. El clasificador, al recibir otras dos instancias de la base de datos (D_k y D_l), utiliza este conocimiento para analizar las instancias de datos en D_k y D_l y realizar predicciones sobre la clasificación de los elementos de D_k y D_l .

Este enfoque general se aplica a todos los enfoques propuestos de coincidencia de esquemas basados en el aprendizaje. La diferencia radica en el tipo de aprendiz que utilizan y cómo ajustan su comportamiento para la coincidencia de esquemas. Algunos han utilizado redes neuronales (p. ej., SEMINT), otros han empleado aprendices/clasificadores bayesianos ingenuos (Autoplex, LSD) y árboles de decisión. No se detallan estas técnicas de aprendizaje.

7.1.2.5 Enfoques de emparejamiento combinado

Las técnicas de emparejamiento individuales que hemos considerado hasta ahora tienen sus ventajas y desventajas. Cada una puede ser más adecuada para emparejar ciertos casos.

Por lo tanto, un algoritmo o metodología de emparejamiento “completo” generalmente necesita utilizar más de un comparador individual.

Existen dos formas posibles de combinar comparadores: híbridos y compuestos. Los algoritmos híbridos combinan varios comparadores en un solo algoritmo.

En otras palabras, los elementos de dos esquemas se pueden comparar utilizando una serie de comparadores de elementos (por ejemplo, coincidencia de cadenas y coincidencia de tipos de datos) y/o comparadores estructurales dentro de un algoritmo para determinar su similitud general.

Los lectores atentos habrán notado que, al analizar los algoritmos de coincidencia basados en restricciones que se centraban en la coincidencia estructural, adoptamos un enfoque híbrido, ya que se basaban en una determinación inicial de la similitud de, por ejemplo, los nodos hoja mediante un comparador de elementos, y estos valores de similitud se utilizaban posteriormente en la coincidencia estructural. Los algoritmos compuestos, por otro lado, aplican cada comparador a los elementos de los dos esquemas (o dos instancias) individualmente, obteniendo puntuaciones de similitud individuales, y luego aplican un método para combinar estas puntuaciones de similitud. Más precisamente, si $s_i(C_k, C_m)$ es la puntuación de similitud utilizando el comparador i

$i (i = 1, \dots, q)$ sobre dos conceptos C_j del esquema k y C_l del esquema m , la similitud compuesta de ambos conceptos se obtiene mediante $s(C_k C_m) = f(s_1, \dots, s_q)$, donde f es la función que combina las puntuaciones de similitud. Esta función puede ser tan simple como promedio, máximo o mínimo, o una adaptación de funciones de agregación de clasificación más complejas, que se analizarán con más detalle en la sección 7.2.

Se ha propuesto un enfoque compuesto en los sistemas LSD e iMAP para gestionar coincidencias 1:1 y N:M, respectivamente.

7.1.3 Integración de esquemas

Una vez realizada la correspondencia de esquemas, se han identificado las correspondencias entre los distintos LCS. El siguiente paso es crear el GCS, lo que se conoce como integración de esquemas. Como se indicó anteriormente, este paso solo es necesario si no se ha definido previamente un GCS y se realizó la correspondencia en cada LCS. Si el GCS se definió previamente, el paso de correspondencia determinaría las correspondencias entre este y cada uno de los LCS, y no sería necesario el paso de integración. Si el GCS se crea como resultado de la integración de los LCS basándose en las correspondencias identificadas durante la correspondencia de esquemas, entonces, como parte de la integración, es importante identificar las correspondencias entre el GCS y los LCS. Si bien se han desarrollado herramientas para facilitar el proceso de integración, la participación humana es esencial.

Ejemplo 7.6. Existen varias integraciones posibles de los dos ejemplos de LCS que hemos analizado. La Figura 7.9 muestra un posible GCS que puede generarse mediante la integración de esquemas. Lo utilizaremos en el resto de este capítulo.

Las metodologías de integración se pueden clasificar como mecanismos binarios o n-arios en función de la forma en que se manejan los esquemas locales en la primera fase (Fig. 7.10). Las metodologías de integración binaria implican la manipulación de dos esquemas a la vez. Esto puede ocurrir de forma escalonada (en escalera) (Fig. 7.11a), donde se crean esquemas intermedios para su integración con esquemas posteriores, o de forma puramente binaria (Fig. 7.11b), donde cada esquema se integra con otro, creando un esquema intermedio para su integración con otros esquemas intermedios.

```

EMP(E#, ENAME, TITLE, CITY)
PAY(TITLE, SAL)
PR(P#, PNAME, BUDGET, LOC)
CL(CNAME, ADDR, CT#, P#)
WORKS(E#, P#, RESP, DUR)

```

Fig. 7.9 Ejemplo de GCS integrado (EMP es empleado, PR es proyecto, CL es cliente)

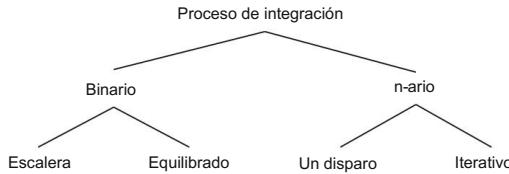


Figura 7.10 Taxonomía de las metodologías de integración

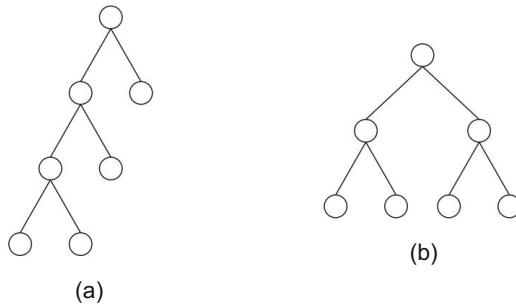


Fig. 7.11 Métodos de integración binaria. (a) Paso a paso. (b) Binario puro.

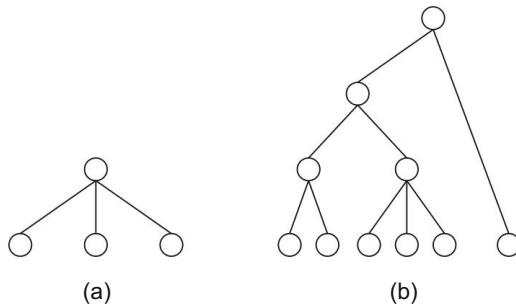


Fig. 7.12 Métodos de integración N-arios. (a) Una pasada. (b) Iterativo.

Los mecanismos de integración N-arios integran más de dos esquemas en cada iteración.

La integración en una sola pasada (Fig. 7.12a) ocurre cuando todos los esquemas se integran simultáneamente, generando el esquema conceptual global tras una iteración. Las ventajas de este enfoque incluyen la disponibilidad de información completa sobre todas las bases de datos en el momento de la integración. No existe una prioridad implícita para el orden de integración de los esquemas, y las compensaciones, como la mejor representación de los elementos de datos o la estructura más comprensible, pueden realizarse entre todos los esquemas en lugar de entre unos pocos. Las dificultades de este enfoque incluyen una mayor complejidad y la dificultad de automatización.

La integración iterativa n-aria (Fig. 7.12b) ofrece mayor flexibilidad (normalmente, hay más información disponible) y es más general (el número de esquemas puede variar según las preferencias del integrador). Los enfoques binarios son un caso especial de

Iterativos n-arios. Reducen la complejidad potencial de la integración y facilitan el uso de técnicas de automatización, ya que el número de esquemas a considerar en cada paso es más manejable. La integración mediante un proceso n-ario permite al integrador realizar operaciones en más de dos esquemas. Por razones prácticas, la mayoría de los sistemas utilizan la metodología binaria, pero varios investigadores prefieren el enfoque n-ario porque se dispone de información completa.

7.1.4 Mapeo de esquemas

Una vez que se define un GCS (o esquema mediado), es necesario identificar cómo se pueden mapear los datos de cada una de las bases de datos locales (fuente) a GCS (destino) mientras se preserva la consistencia semántica (tal como lo definen tanto la fuente como el destino).

Aunque la coincidencia de esquemas ha identificado las correspondencias entre los LCS y los GCS, es posible que no haya identificado explícitamente cómo obtener la base de datos global a partir de las locales. De esto se trata el mapeo de esquemas.

En el caso de los almacenes de datos, las asignaciones de esquemas se utilizan para extraer datos explícitamente de las fuentes y traducirlos al esquema del almacén de datos para su posterior procesamiento. En el caso de los sistemas de integración de datos, estas asignaciones se utilizan en la fase de procesamiento de consultas, tanto por el procesador de consultas como por los wrappers (véase la sección 7.2).

Hay dos aspectos relacionados con el mapeo de esquemas que estudiaremos: la creación y el mantenimiento de mapeos. La creación de mapeos consiste en crear consultas explícitas que mapean datos de una base de datos local a la global. El mantenimiento de mapeos consiste en detectar y corregir inconsistencias en los mapeos resultantes de la evolución del esquema. Los esquemas fuente pueden sufrir cambios estructurales o semánticos que invalidan los mapeos. El mantenimiento de mapeos se encarga de detectar mapeos defectuosos y reescribirlos (automáticamente) para lograr la coherencia semántica con el nuevo esquema y la equivalencia semántica con el mapeo actual.

7.1.4.1 Creación de mapas

La creación de mapas comienza con un LCS de origen, el GCS de destino y un conjunto de coincidencias de esquema M. Esto genera un conjunto de consultas que, al ejecutarse, crean instancias de datos GCS a partir de los datos de origen. En los almacenes de datos, estas consultas se ejecutan para crear el almacén de datos (base de datos global), mientras que en los sistemas de integración de datos, se utilizan en sentido inverso durante el procesamiento de consultas (Sección 7.2).

Para concretar esto, consultemos la representación relacional canónica que hemos adoptado. El LCS fuente en cuestión consiste en un conjunto de relaciones Origen = {O₁, ..., O_m}, el GCS consiste en un conjunto de relaciones globales (o de destino) Destino = {T₁, ..., T_n}, y M consiste en un conjunto de reglas de coincidencia de esquema, como se define en la sección 7.1.2. Buscamos una manera de generar, para cada T_k, una consulta.

Q_k que se define en un subconjunto (posiblemente apropiado) de las relaciones en Source de modo que, cuando se ejecuta, generará datos para Tk a partir de las relaciones de origen.

Esto se puede lograr iterativamente considerando cada Tk por turno. Comienza con $M_k = M$ (M_k es el conjunto de reglas que solo se aplican a los atributos de Tk) y lo divide en subconjuntos $\{M_1, \dots, M_s\}$ que especifican una posible mapeo de Tk a calcular los valores de Tk. Cada M_j ejecutado generaría algunos de los datos de Tk. La unión k se puede asignar a una consulta q_k que cuando de todas estas consultas proporciona $Q_k (= \bigcup_j q_j)$ que buscamos.

k

El algoritmo se desarrolla en cuatro pasos que se describen a continuación. No considera los valores de similitud en las reglas. Se puede argumentar que los valores de similitud se utilizarían en las etapas finales del proceso de emparejamiento para finalizar las correspondencias, por lo que su uso durante el mapeo es innecesario. Además, al llegar a esta fase del proceso de integración, la preocupación es cómo mapear los datos de la relación de origen (LCS) con los datos de la relación de destino (GCS). En consecuencia, las correspondencias no son equivalencias simétricas (\approx), sino mapeos (\rightarrow): los atributos de (posiblemente múltiples) relaciones de origen se mapean a un atributo de una relación de destino (es decir, $(O_i.attr\ ibute_k, O_j.attr\ ibute_l) \rightarrow T_w.attr\ ibute_z$).

Ejemplo 7.7 Para demostrar el algoritmo, utilizaremos una base de datos de ejemplo diferente a la que hemos estado utilizando, ya que no incorpora todas las complejidades que queremos demostrar. En su lugar, utilizaremos el siguiente ejemplo abstracto.

Relaciones de origen (LCS):

- O1(A1, A2)
- O2(B1, B2, B3)
- O3(C1, C2, C3)
- O4(D1, D2)

Relación objetivo (GCS):

- T(W1, W2, W3, W4)

Consideraremos solo una relación en GCS ya que el algoritmo itera sobre las relaciones de destino una a la vez; esto es suficiente para demostrar el funcionamiento del algoritmo.

Las relaciones de clave externa entre los atributos son las siguientes:

Clave externa	Se refiere a
A1	B1
A2	B1
C1	B1

Supongamos que se han descubierto las siguientes coincidencias para los atributos de la relación T (que conforman MT). En los ejemplos subsiguientes, no nos ocuparemos de los predicados, por lo que no se especifican explícitamente.

$r1 = A1 \rightarrow W1, p$
 $r2 = A2 \rightarrow W2, p$
 $r3 = B2 \rightarrow W4, p$
 $r4 = B3 \rightarrow W3, p$
 $r5 = C1 \rightarrow W1, p$
 $r6 = C2 \rightarrow W2, p$
 $r7 = D1 \rightarrow W4, p$

En el primer paso, M_k (correspondiente a T_k) se divide en sus subconjuntos

$\{M_{1k}, \dots, M_{nk}\}$ tal que cada M_j contiene como máximo una coincidencia para cada atributo de T_k . Estos se denominan conjuntos de candidatos potenciales, algunos de los cuales pueden estar completos, ya que incluyen una coincidencia para cada atributo de T_k , pero otros pueden no serlo.

Las razones para considerar conjuntos incompletos son dos. En primer lugar, puede darse el caso de que No se encuentra ninguna coincidencia para uno o más atributos de la relación de destino (es decir, ninguno de los conjuntos de coincidencias están completos). En segundo lugar, para esquemas de bases de datos grandes y complejos, Puede tener sentido construir el mapeo de forma iterativa para que el diseñador especifique las asignaciones incrementales.

Ejemplo 7.8 MT se divide en cincuenta y tres subconjuntos (es decir, candidatos potenciales) conjuntos). Los primeros ocho están completos, mientras que el resto no. Mostramos algunos de estos a continuación. Para facilitar la lectura, las reglas completas se enumeran en el orden de los atributos de destino a los que se asignan (por ejemplo, la tercera regla en M_1 es $r4$, porque Esta regla se asigna al atributo W_3):

$M1_T = \{r1, r2, r4, r3\}$	$M2_T = \{r1, r2, r4, r7\}$
$M3_T = \{r1, r6, r4, r3\}$	$M4_T = \{r1, r6, r4, r7\}$
$M5_T = \{r5, r2, r4, r3\}$	$M6_T = \{r5, r2, r4, r7\}$
$M7_T = \{r5, r6, r4, r3\}$	$M8_T = \{r5, r6, r4, r7\}$
$M9_T = \{r1, r2, r3\}$	$M10_T = \{r1, r2, r4\}$
$M11_T = \{r1, r3, r4\}$	$M12_T = \{r2, r3, r4\}$
$M13_T = \{r1, r3, r6\}$	$M14_T = \{r3, r4, r6\}$
...	...
$M47_T = \{r1\}$	$M48_T = \{r2\}$
$M49_T = \{r3\}$	$M50_T = \{r4\}$
$M51_T = \{r5\}$	$M52_T = \{r6\}$
$M53_T = \{r7\}$	

En el segundo paso, el algoritmo analiza cada conjunto de candidatos potenciales M_j a vea si se puede producir una consulta "buena" para ello. Si todas las coincidencias en M_j asignan valores de una única relación de origen a T_k , entonces es fácil generar una consulta correspondiente a M_j . De particular preocupación son las coincidencias que requieren acceso a múltiples relaciones de origen. En este caso, el algoritmo verifica si hay una conexión referencial entre estas relaciones a través de claves externas (es decir, si hay una ruta de unión a través de las relaciones de origen). Si no la hay, entonces el conjunto de candidatos potenciales se elimina de la consideración posterior. En caso de que haya múltiples rutas de unión a través de relaciones de clave externa, el algoritmo busca aquellas rutas que producirán el mayor número de tuplas (es decir, la diferencia estimada en tamaño de las uniones externas e internas es la más pequeña). Si hay múltiples rutas de este tipo, entonces el diseñador de la base de datos debe involucrarse en la selección de una (herramientas como Clio, OntoBuilder y otras facilitan este proceso y proporcionan mecanismos para que los diseñadores vean y especifiquen correspondencias). El resultado de este paso es un conjunto M_k de conjuntos candidatos.

Ejemplo 7.9 de M_j . En este ejemplo, no hay reglas asignadas k donde los valores de todos los atributos de T desde una única relación de origen. Entre las que involucran múltiples relaciones de origen, las reglas que involucran O_1, O_2 y O_3 pueden asignarse a consultas "buenas", ya que existen relaciones de clave externa entre ellas. Sin embargo, las reglas que involucran O_4 (es decir, las que incluyen la regla $r7$) no pueden asignarse a una consulta "buena", ya que no existe una ruta de unión de O_4 a las demás relaciones (es decir, cualquier consulta implicaría un producto vectorial, lo cual es costoso). Por lo tanto, estas reglas se eliminan del posible conjunto de candidatos M_6 . Considerando solo los conjuntos completos, M_2 ,

k, k^M_4 , y M_8 se podan k

Del conjunto. Finalmente, el conjunto candidato (M_k) contiene treinta y cinco reglas (se recomienda a los lectores verificar esto para comprender mejor el algoritmo).

En el tercer paso, el algoritmo busca una cobertura de los conjuntos candidatos M_k . La cobertura C_k M_k es un conjunto de conjuntos candidatos tal que cada coincidencia en M_k aparece en C_k al menos una vez. El objetivo de determinar una cobertura es que esta tenga en cuenta todas las coincidencias y, por lo tanto, sea suficiente para generar la relación objetivo T_k . Si existen múltiples coberturas (una coincidencia puede participar en varias coberturas), se ordenan en orden creciente de conjuntos candidatos. Cuanto menor sea el número de conjuntos candidatos en la cobertura, menor será el número de consultas que se generarán en el siguiente paso; esto mejora la eficiencia de las asignaciones generadas. Si existen múltiples coberturas con la misma clasificación, se ordenan en orden decreciente según el número total de atributos objetivo únicos utilizados en los conjuntos candidatos que constituyen la cobertura. El objetivo de esta clasificación es que las coberturas con mayor número de atributos generen menos valores nulos en el resultado. En esta etapa, puede ser necesario consultar al diseñador para elegir entre las coberturas clasificadas.

Ejemplo 7.10. Observe primero que tenemos seis reglas que definen coincidencias en M_k que debemos considerar, ya que se han eliminado las reglas M_j que incluyen la regla $r7$. Existe una gran cantidad de posibles coberturas; comencemos con las que involucran a M_1 para k a demostrar el algoritmo:

$$C_1 = \{\{r1, r2, r4, r3\}, \{r1, r6, r4, r3\}, \{r2\} \}$$

M1 M3 M48

$$C_2 = \{\{r1, r2, r4, r3\}, \{r5, r2, r4, r3\}, \{r6\} \}$$

M1 M5 M50

$$C_3 = \{\{r1, r2, r4, r3\}, \{r5, r6, r4, r3\} \}$$

M1 M7

$$C_4 = \{\{r1, r2, r4, r3\}, \{r5, r6, r4\} \}$$

M1 M12

$$C_5 = \{\{r1, r2, r4, r3\}, \{r5, r6, r3\} \}$$

M1 M19

$$C_6 = \{\{r1, r2, r4, r3\}, \{r5, r6\} \}$$

M1 M32

En este punto observamos que las cubiertas constan de dos o tres candidatos.

conjuntos. Dado que el algoritmo prefiere aquellos con menos conjuntos candidatos, solo necesitamos centrarse en las que involucran dos conjuntos. Además, entre estas portadas, destacamos que la El número de atributos objetivo en los conjuntos de candidatos difiere. Dado que el algoritmo prefiere cubiertas con el mayor número de atributos objetivo en cada conjunto de candidatos, C3 T es el cubierta preferida

Tenga en cuenta que debido a las dos heurísticas empleadas por el algoritmo, las únicas coberturas que tenemos son: Es necesario considerar aquellos que involucran M1 Y0, M3 Y0, M5 Y0, y M7 T. Se pueden encontrar cubiertas similares definido involucrando M3 Y0, M5 Y0, y M7 T; lo dejamos como ejercicio. En el resto, Supondremos que el diseñador ha optado por utilizar C3 T como la cubierta preferida.

El paso final del algoritmo construye una consulta qj k para cada uno de los conjuntos candidatos En la portada seleccionada en el paso anterior. La unión de todas estas consultas (UNION) ALL) da como resultado el mapeo final para la relación Tk en el GCS.

Consulta qj k Se construye de la siguiente manera:

- La cláusula SELECT incluye todas las correspondencias (c) en cada una de las reglas (ri k) en Mj k .
- La cláusula FROM incluye todas las relaciones de origen mencionadas en ri k y en la unión rutas determinadas en el paso 2 del algoritmo.
- La cláusula WHERE incluye el conjunto de todos los predicados (p) en ri k y todos se unen predicados determinados en el paso 2 del algoritmo.
- Si ri k contiene una función agregada ya sea en c o en p
 - GROUP BY se utiliza sobre atributos (o funciones sobre atributos) en el Cláusula SELECT que no están dentro del agregado;

- Si el agregado está en la correspondencia c, se agrega a SELECT, de lo contrario (es decir, si el agregado está en el predicado p) se crea una cláusula HAVING con el agregado.

Ejemplo 7.11 Dado que en el Ejemplo 7.10 decidimos usar la asignación de cobertura T para la final C3, necesitamos generar dos consultas: q1 y q7, correspondientes a M1, respectivamente. T y $M7$. Para facilitar la presentación, volvemos a enumerar las reglas:

$$r1 = A1 \rightarrow W1, p$$

$$r2 = A2 \rightarrow W2, p$$

$$r3 = B2 \rightarrow W4, p$$

$$r4 = B3 \rightarrow W3, p$$

$$r5 = C1 \rightarrow W1, p$$

$$r6 = C2 \rightarrow W2, p$$

Las dos consultas son las siguientes:

q1: SELECCIONE A1, A2, B2, B3 DE O1, O2
 DONDE p1 Y O1.A2 =
 O2.B1

q7: SELECCIONE B2, B3, C1, C2 DE O2, O3
 DONDE p3 Y p4 Y p5 Y
 p6 Y O3.c1 = O2.B1

Por lo tanto, la consulta final Qk para la relación objetivo T se convierte en q1 UNIÓN TODOS q7.

El resultado de este algoritmo, tras su aplicación iterativa a cada relación de destino Tk, es un conjunto de consultas $Q = \{Q_k\}$ que, al ejecutarse, generan datos para las relaciones GCS. Por lo tanto, el algoritmo genera mapeos GAV entre esquemas relacionales. Cabe recordar que GAV define un GCS como una vista de los LCS, y eso es exactamente lo que hace el conjunto de consultas de mapeo. El algoritmo considera la semántica del esquema fuente, ya que considera las relaciones de clave externa para determinar las consultas que se generarán. Sin embargo, no considera la semántica del destino, por lo que no se garantiza que las tuplas generadas por la ejecución de las consultas de mapeo satisfagan la semántica del destino. Esto no supone un problema importante cuando el GCS se integra desde los LCS; sin embargo, si el GCS se define independientemente de los LCS, sí resulta problemático.

Es posible extender el algoritmo para gestionar tanto la semántica de destino como la de origen. Esto requiere considerar las dependencias generadoras de tuplas entre esquemas. En otras palabras, es necesario generar mapeos GLAV. Un GLAV

El mapeo, por definición, no es simplemente una consulta sobre las relaciones de origen; es una relación entre una consulta sobre las relaciones de origen (es decir, LCS) y una consulta sobre las relaciones de destino (es decir, GCS). Seamos más precisos. Considere una coincidencia de esquema v que especifica una correspondencia entre el atributo A de una relación LCS de origen R y el atributo B de una relación GCS de destino T (en la notación que usamos en esta sección tenemos $v = RA \approx TB, p, s$). Luego, la consulta de origen especifica cómo recuperar RA y la consulta de destino especifica cómo obtener TB. El mapeo GLAV, entonces, es una relación entre estas dos consultas.

Esto se puede lograr partiendo de un esquema de origen, un esquema de destino y M, y descubriendo asignaciones que satisfagan tanto la semántica del esquema de origen como la del de destino. Este algoritmo también es más potente que el que analizamos en esta sección, ya que puede manejar estructuras anidadas comunes en XML, bases de datos de objetos y sistemas relacionales anidados.

El primer paso para descubrir todas las asignaciones basadas en correspondencias de esquemas es la traducción semántica, que busca interpretar las coincidencias de esquemas en M de forma coherente con la semántica de los esquemas de origen y destino, tal como se captura mediante la estructura del esquema y las restricciones referenciales (clave externa). El resultado es un conjunto de asignaciones lógicas, cada una de las cuales captura las decisiones de diseño (semántica) realizadas en los esquemas de origen y destino. Cada asignación lógica corresponde a una relación del esquema de destino. El segundo paso es la traducción de datos, que implementa cada asignación lógica como una regla que puede traducirse en una consulta que, al ejecutarse, crearía una instancia del elemento de destino.

La traducción semántica toma como entradas los esquemas fuente y destino.

Objetivo y M y realiza los dos pasos siguientes:

- Examina la semántica intraesquemática dentro del origen y el destino por separado y produce para cada uno un conjunto de relaciones lógicas que son semánticamente consistentes. • Luego interpreta las correspondencias interesquemáticas M en el contexto de las relaciones lógicas generadas en el paso 1 y produce un conjunto de consultas en Q que son semánticamente consistentes con el destino.

7.1.4.2 Mantenimiento de mapas

En entornos dinámicos donde los esquemas evolucionan con el tiempo, las asignaciones de esquemas pueden volverse inválidas como resultado de cambios estructurales o de restricciones de los esquemas. Por lo tanto, la detección de asignaciones de esquemas no válidas o inconsistentes y la adaptación de dichas asignaciones de esquemas a nuevas estructuras/restricciones de esquema son importantes.

En general, la detección automática de asignaciones de esquemas inválidas o inconsistentes es deseable a medida que aumenta la complejidad de los esquemas y el número de asignaciones utilizadas en las aplicaciones de bases de datos. Asimismo, la adaptación (semi)automática de las asignaciones a los cambios de esquema también es un objetivo. Cabe destacar que la adaptación automática de las asignaciones de esquemas no es lo mismo que la coincidencia automática de esquemas.

La adaptación del esquema tiene como objetivo resolver correspondencias semánticas utilizando cambios conocidos en la semántica intraesquemática, la semántica en asignaciones existentes y la semántica detectada.

Inconsistencias (resultantes de cambios de esquema). La correspondencia de esquemas debe adoptar un enfoque mucho más "desde cero" para generar asignaciones de esquemas y no tiene la capacidad (ni el lujo) de incorporar dicho conocimiento contextual.

Detección de asignaciones no válidas

En general, la detección de asignaciones no válidas resultantes de cambios de esquema puede realizarse de forma proactiva o reactiva. En entornos de detección proactiva, se comprueban las asignaciones de esquema para detectar inconsistencias en cuanto un usuario realiza cambios. Se asume (o requiere) que el sistema de mantenimiento de asignaciones esté completamente al tanto de todos los cambios de esquema en cuanto se realizan. El sistema ToMAS, por ejemplo, espera que los usuarios realicen cambios de esquema a través de sus propios editores de esquemas, lo que permite al sistema detectar inmediatamente cualquier cambio.

Una vez que se han detectado cambios en el esquema, se pueden detectar asignaciones no válidas realizando una traducción semántica de las asignaciones existentes utilizando las relaciones lógicas del esquema actualizado.

En entornos de detección reactiva, el sistema de mantenimiento de mapeos desconoce cuándo y qué cambios de esquema se realizan. Para detectar mapeos de esquema no válidos en este entorno, se prueban los mapeos periódicamente mediante consultas a las fuentes de datos y la traducción de los datos resultantes utilizando los mapeos existentes.

Las asignaciones no válidas se determinan luego en función de los resultados de estas pruebas de asignación.

Un método alternativo propuesto consiste en utilizar técnicas de aprendizaje automático para detectar asignaciones no válidas (como en el sistema Maveric). Se propone construir un conjunto de sensores entrenados (similar a múltiples aprendices en la coincidencia de esquemas) para detectar asignaciones no válidas. Entre estos sensores se incluyen sensores de valor para monitorizar las características de distribución de los valores de las instancias objetivo, sensores de tendencia para monitorizar la tasa media de modificación de los datos, y sensores de diseño y restricción que monitorizan los datos traducidos en relación con la sintaxis y la semántica esperadas del esquema objetivo. Se calcula entonces una combinación ponderada de los hallazgos de cada sensor, donde también se aprenden las ponderaciones. Si el resultado combinado indica cambios y las pruebas de seguimiento sugieren que esto podría ser así, se genera una alerta.

Adaptación de asignaciones no válidas

Una vez que se detectan asignaciones de esquema inválidas, deben adaptarse a los cambios de esquema y validarse nuevamente. Se han propuesto diversos enfoques de adaptación de mapeo de alto nivel. Estos pueden describirse, en términos generales, como enfoques de reglas fijas que definen una regla de reasignación para cada tipo de cambio de esquema esperado; enfoques de puenteo de mapas que comparan el esquema original S con el esquema actualizado S y generan nuevas asignaciones de S a S, además de las existentes; y enfoques de reescritura semántica, que aprovechan la información semántica codificada en asignaciones, esquemas y cambios semánticos existentes realizados en los esquemas para proponer un mapa.

Reescrituras que producen datos objetivo semánticamente consistentes. En la mayoría de los casos, es posible realizar múltiples reescrituras, lo que requiere una clasificación de los candidatos para su presentación a los usuarios que toman la decisión final (basada en la semántica a nivel de escenario o de negocio, no codificada en esquemas ni mapeos).

Podría decirse que una reasignación completa de esquemas (es decir, desde cero, mediante técnicas de coincidencia de esquemas) es otra alternativa a la adaptación de mapas. Sin embargo, en la mayoría de los casos, la reescritura de mapas es más económica que la regeneración de mapas, ya que la reescritura permite aprovechar el conocimiento codificado en los mapas existentes para evitar el cálculo de mapas que el usuario rechazaría de todos modos (y para evitar mapeos redundantes).

7.1.5 Limpieza de datos

Siempre pueden ocurrir errores en las bases de datos de origen, lo que requiere una limpieza para responder correctamente a las consultas de los usuarios. La limpieza de datos es un problema que surge tanto en almacenes de datos como en sistemas de integración de datos, pero en contextos diferentes. En los almacenes de datos, donde los datos se extraen de bases de datos operativas locales y se materializan en una base de datos global, la limpieza se realiza a medida que se crea dicha base de datos. En el caso de los sistemas de integración de datos, la limpieza de datos es un proceso que debe realizarse durante el procesamiento de consultas, cuando se devuelven datos de las bases de datos de origen.

Los errores que están sujetos a la limpieza de datos generalmente se pueden desglosar en problemas a nivel de esquema o a nivel de instancia. Los problemas a nivel de esquema pueden surgir en cada LCS individual debido a violaciones de restricciones explícitas e implícitas. Por ejemplo, los valores de los atributos pueden estar fuera del rango de sus dominios (p. ej., 14.^o mes o valor salarial negativo), los valores de los atributos pueden violar dependencias implícitas (p. ej., el valor del atributo de edad puede no corresponder al valor que se calcula como la diferencia entre la fecha actual y la fecha de nacimiento), la unicidad de los valores de los atributos puede no cumplirse y las restricciones de integridad referencial pueden violarse. Además, en el entorno que estamos considerando en este capítulo, las heterogeneidades a nivel de esquema (tanto estructurales como semánticas) entre los LCS que discutimos anteriormente pueden considerarse problemas que necesitan ser resueltos. A nivel de esquema, es claro que los problemas necesitan ser identificados en la etapa de coincidencia de esquemas y solucionados durante la integración de esquemas.

Los errores a nivel de instancia son aquellos que existen a nivel de datos. Por ejemplo, los valores de algunos atributos pueden faltar aunque sean obligatorios, podría haber errores ortográficos y transposiciones de palabras (p. ej., "MD Mary Smith" frente a "Mary Smith, MD") o diferencias en las abreviaturas (p. ej., "J. Doe" en una base de datos de origen, mientras que "JN Doe" en otra), valores incrustados (p. ej., un atributo de dirección agregado que incluye el nombre de la calle, el valor, el nombre de la provincia y el código postal), valores que se colocaron erróneamente en otros campos, valores duplicados y valores contradictorios (el valor del salario aparece como un valor en una base de datos y como otro valor en otra). Para la limpieza a nivel de instancia, el problema radica claramente en la generación de las asignaciones de modo que los datos se limpian mediante la ejecución de las funciones de asignación (consultas).

El enfoque popular para la limpieza de datos ha consistido en definir una serie de operadores que operan sobre esquemas o datos individuales. Estos operadores pueden integrarse en un plan de limpieza de datos. Algunos ejemplos de operadores de esquema son añadir o eliminar columnas de una tabla, reestructurar una tabla combinando columnas o dividiendo una columna en dos, o definir una transformación de esquema más compleja mediante un operador genérico de "mapa" que toma una sola relación y genera una o más. Entre los operadores a nivel de datos se incluyen aquellos que aplican una función a cada valor de un atributo, la fusión de valores de dos atributos en el valor de un solo atributo y su operador de división inverso, un operador de coincidencia que calcula una unión aproximada entre tuplas de dos relaciones, un operador de agrupamiento que agrupa las tuplas de una relación en clústeres, y un operador de fusión de tuplas que divide las tuplas de una relación en grupos y las contrae en una sola tupla mediante la agregación, así como operadores básicos para encontrar duplicados y eliminarlos. Muchos de los operadores a nivel de datos comparan tuplas individuales de dos relaciones (del mismo o diferente esquema) y determinan si representan el mismo hecho. Esto es similar a lo que se hace en la comparación de esquemas, excepto que se realiza a nivel de datos individuales y lo que se considera no son valores de atributos individuales, sino tuplas completas. Sin embargo, las mismas técnicas que estudiamos en la comparación de esquemas (p. ej., el uso de la distancia de edición o el valor de soundex) pueden emplearse en este contexto. Se han propuesto técnicas especiales para gestionar esto eficientemente en el contexto de la limpieza de datos, como la comparación difusa, que calcula una función de similitud para determinar si las dos tuplas son idénticas o razonablemente similares.

Dada la gran cantidad de datos que deben gestionarse, la limpieza a nivel de datos es costosa y la eficiencia es un problema importante. La implementación física de cada uno de los operadores mencionados anteriormente es una preocupación considerable. Si bien la limpieza puede realizarse fuera de línea como un proceso por lotes en el caso de los almacenes de datos, para los sistemas de integración de datos, la limpieza puede requerirse en línea a medida que se recuperan los datos de las fuentes. El rendimiento de la limpieza de datos es, por supuesto, más crítico en estos últimos casos.

7.2 Procesamiento de consultas en múltiples bases de datos

Ahora nos centraremos en la consulta y el acceso a una base de datos integrada obtenida mediante las técnicas descritas en la sección anterior; esto se conoce como el problema de consulta multibase de datos. Como se mencionó anteriormente, muchas de las técnicas de procesamiento distribuido de consultas y optimización que analizamos en el capítulo 4 se aplican a los sistemas multibase de datos, pero existen diferencias importantes. Recordemos que en ese capítulo caracterizamos el procesamiento distribuido de consultas en cuatro pasos: descomposición de consultas, localización de datos, optimización global y optimización local. La naturaleza de los sistemas multibase de datos requiere pasos y técnicas ligeramente diferentes. Los SGBD que los componen pueden ser autónomos y tener diferentes lenguajes de base de datos y capacidades de procesamiento de consultas. Por lo tanto, se necesita una capa de SGBD (véase la figura 1.12) para comunicarse eficazmente con los SGBD que los componen, lo que requiere

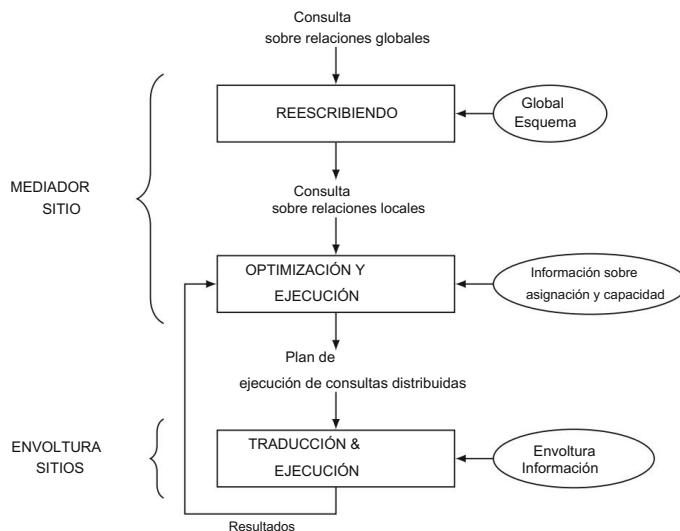


Fig. 7.13 Esquema de capas genérico para el procesamiento de consultas de múltiples bases de datos

Pasos adicionales de procesamiento de consultas (Fig. 7.13). Además, puede haber muchos SGBD que los componen, cada uno con un comportamiento diferente, lo que plantea nuevos requisitos para técnicas de procesamiento de consultas más adaptativas.

7.2.1 Problemas en el procesamiento de consultas de múltiples bases de datos

El procesamiento de consultas en un sistema multibase de datos es más complejo que en un sistema distribuido. DBMS por las siguientes razones:

1. Las capacidades de procesamiento de los SGBD que los componen pueden variar, lo que impide un procesamiento uniforme de las consultas en varios SGBD. Por ejemplo, algunos SGBD admiten consultas SQL complejas con unión y agregación, mientras que otros no. Por lo tanto, el procesador de consultas multibase de datos debe considerar las distintas capacidades del SGBD. Las capacidades de cada componente se registran en el directorio junto con la información de asignación de datos.
2. De igual manera, el costo de procesar consultas puede variar según el sistema de gestión de bases de datos (SGBD), y la capacidad de optimización local de cada SGBD puede ser muy diferente. Esto aumenta la complejidad de las funciones de costo que deben evaluarse.
3. Los modelos de datos y lenguajes de los DBMS componentes pueden ser bastante diferentes, por ejemplo, relacionales, orientados a objetos, semiestructurados, etc. Esto crea dificultades a la hora de traducir consultas de múltiples bases de datos a los DBMS componentes y de integrar resultados heterogéneos.

4. Dado que un sistema multibase de datos permite el acceso a DBMS muy diferentes que pueden tener diferente rendimiento y comportamiento, las técnicas de procesamiento de consultas distribuidas deben adaptarse a estas variaciones.

La autonomía de los componentes de los SGBD plantea problemas. Esta autonomía se puede definir en tres dimensiones principales: comunicación, diseño y ejecución.

La autonomía de comunicación significa que un componente del sistema de gestión de bases de datos (SGBD) se comunica con otros a su propia discreción y, en particular, puede finalizar sus servicios en cualquier momento. Esto requiere técnicas de procesamiento de consultas que toleren la indisponibilidad del sistema. La cuestión es cómo responde el sistema a las consultas cuando un componente del sistema no está disponible desde el principio o se apaga durante la ejecución de la consulta. La autonomía de diseño puede restringir la disponibilidad y la precisión de la información de costes necesaria para la optimización de consultas. La dificultad para determinar las funciones de costes locales es un aspecto importante. La autonomía de ejecución de los sistemas multibase de datos dificulta la aplicación de algunas de las estrategias de optimización de consultas que analizamos en capítulos anteriores. Por ejemplo, la optimización basada en semijoins de uniones distribuidas puede resultar difícil si las relaciones de origen y destino residen en diferentes SGBD de componentes, ya que, en este caso, la ejecución de semijoins de una unión se traduce en tres consultas: una para recuperar los valores de los atributos de unión de la relación de destino y enviarlos al SGBD de la relación de origen, la segunda para realizar la unión en la relación de origen y la tercera para realizar la unión en el SGBD de la relación de destino. El problema surge porque la comunicación con los componentes DBMS se produce en un nivel alto de la API del DBMS.

Además de estas dificultades, la arquitectura de un sistema multibase de datos distribuido plantea ciertos desafíos. La arquitectura mostrada en la Fig. 1.12 indica una complejidad adicional. En los SGBD distribuidos, los procesadores de consultas solo deben gestionar la distribución de datos entre múltiples sitios. En un entorno multibase de datos distribuido, por otro lado, los datos se distribuyen no solo entre sitios, sino también entre múltiples bases de datos, cada una gestionada por un SGBD autónomo. Por lo tanto, mientras que en un SGBD distribuido hay dos partes que cooperan en el procesamiento de consultas (el sitio de control y los sitios locales), el número de partes aumenta a tres en el caso de un SGBD distribuido: la capa del SGBD en el sitio de control (es decir, el mediador) recibe la consulta global, las capas del SGBD en los sitios (es decir, los wrappers) participan en el procesamiento de la consulta y, en última instancia, los SGBD que la componen optimizan y ejecutan la consulta.

7.2.2 Arquitectura de procesamiento de consultas de múltiples bases de datos

La mayor parte del trabajo sobre el procesamiento de consultas multibase de datos se ha realizado en el contexto de la arquitectura mediador/encapsulador (véase la Fig. 1.13). En esta arquitectura, cada base de datos componente tiene un encapsulador asociado que exporta información sobre el esquema de origen, los datos y las capacidades de procesamiento de consultas. Un mediador centraliza la información proporcionada por los encapsuladores en una vista unificada de todos los datos disponibles (almacenados).

en un diccionario de datos global) y procesa las consultas mediante los wrappers para acceder a los SGBD que las componen. El modelo de datos que utiliza el mediador puede ser relacional, orientado a objetos o incluso semiestructurado. En este capítulo, para mantener la coherencia con los capítulos anteriores sobre procesamiento distribuido de consultas, seguimos utilizando el modelo relacional, que es suficiente para explicar las técnicas de procesamiento de consultas multibase de datos.

La arquitectura mediador/encapsulador ofrece varias ventajas. En primer lugar, sus componentes especializados permiten gestionar por separado las distintas necesidades de los distintos tipos de usuarios. En segundo lugar, los mediadores suelen especializarse en un conjunto relacionado de bases de datos de componentes con datos similares, exportando así esquemas y semántica relacionados con un dominio específico. La especialización de los componentes da lugar a un sistema distribuido flexible y extensible. En particular, permite la integración fluida de distintos datos almacenados en componentes muy diversos, desde sistemas de gestión de bases de datos relacionales completos hasta archivos simples.

Asumiendo la arquitectura mediador/encapsulador, podemos analizar las distintas capas involucradas en el procesamiento de consultas en sistemas multibase de datos distribuidos, como se muestra en la Fig. 7.13. Como antes, asumimos que la entrada es una consulta sobre relaciones globales expresadas en cálculo relacional. Esta consulta se plantea sobre relaciones globales (distribuidas), lo que significa que la distribución y la heterogeneidad de los datos quedan ocultas. El procesamiento de consultas multibase de datos consta de tres capas principales. Esta estratificación es similar a la del procesamiento de consultas en sistemas de gestión de bases de datos distribuidos homogéneos (véase la Fig. 4.2). Sin embargo, al no haber fragmentación, no se requiere la capa de localización de datos.

Las dos primeras capas asignan la consulta de entrada a un plan de ejecución de consultas distribuidas (QEP) optimizado. Realizan las funciones de reescritura, optimización y ejecución de consultas. Las dos primeras capas son ejecutadas por el mediador y utilizan metainformación almacenada en el directorio global (esquema global, asignación e información de capacidad). La reescritura de consultas transforma la consulta de entrada en una consulta sobre relaciones locales, utilizando el esquema global. Cabe recordar que existen dos enfoques principales para la integración de bases de datos: global como vista (GAV) y local como vista (LAV). Por lo tanto, el esquema global proporciona las definiciones de vista (es decir, las asignaciones entre las relaciones globales y locales almacenadas en las bases de datos componentes) y la consulta se reescribe utilizando las vistas.

La reescritura puede realizarse a nivel de cálculo relacional o álgebra. En este capítulo, utilizaremos una forma generalizada de cálculo relacional denominada Datalog, ideal para dicha reescritura. Por lo tanto, existe un paso adicional de traducción de cálculo a álgebra, similar al paso de descomposición en los SGBD distribuidos homogéneos.

La segunda capa optimiza las consultas y realiza (parte de) su ejecución considerando la asignación de las relaciones locales y las diferentes capacidades de procesamiento de consultas de los SGBD componentes exportados por los envoltorios. La información de asignación y capacidad que utiliza esta capa también puede contener información de costes heterogénea. El QEP distribuido generado por esta capa agrupa en subconsultas las operaciones que pueden realizar los SGBD componentes y los envoltorios.

De manera similar a los DBMS distribuidos, la optimización de consultas puede ser estática o dinámica. Sin embargo, la falta de homogeneidad en los sistemas de múltiples bases de datos (por ejemplo, algunos componentes

Los SGBD pueden presentar retrasos inesperados en las respuestas, lo que hace que la optimización de consultas dinámicas sea más crítica. En el caso de la optimización dinámica, puede haber llamadas posteriores a esta capa tras la ejecución de la capa subsiguiente, como lo ilustra la flecha que muestra los resultados de las capas de traducción y ejecución. Finalmente, esta capa integra los resultados de los diferentes envoltorios para proporcionar una respuesta unificada a la consulta del usuario. Esto requiere la capacidad de ejecutar algunas operaciones con los datos provenientes de los envoltorios. Dado que los envoltorios pueden ofrecer capacidades de ejecución muy limitadas (por ejemplo, en el caso de SGBD con componentes muy simples), el mediador debe proporcionar todas las capacidades de ejecución para soportar la interfaz del mediador.

La tercera capa realiza la traducción y ejecución de consultas utilizando los envoltorios.

Luego, devuelve los resultados al mediador, que puede integrar los resultados de diferentes wrappers y ejecutarlos posteriormente. Cada wrapper mantiene un esquema que incluye el esquema de exportación local y la información de mapeo para facilitar la traducción de la subconsulta de entrada (un subconjunto del QEP), expresada en un lenguaje común, al lenguaje del SGBD del componente. Una vez traducida la subconsulta, el SGBD del componente la ejecuta y el resultado local se traduce de nuevo al formato común.

La información del contenedor describe cómo se pueden realizar las asignaciones desde/hacia los esquemas locales participantes y el esquema global. Permite conversiones entre componentes de la base de datos de diferentes maneras. Por ejemplo, si el esquema global representa las temperaturas en grados Fahrenheit, pero una base de datos participante utiliza grados Celsius, la información del contenedor debe contener una fórmula de conversión para proporcionar la presentación correcta al usuario global y a las bases de datos locales. Si la conversión se realiza entre tipos y las fórmulas simples no pueden realizar la traducción, se pueden usar tablas de asignaciones completas en la información del contenedor almacenada en el directorio.

7.2.3 Reescritura de consultas mediante vistas

La reescritura de consultas reformula la consulta de entrada expresada en relaciones globales en una consulta en relaciones locales. Utiliza el esquema global, que describe en términos de vistas las correspondencias entre las relaciones globales y locales. Por lo tanto, la consulta debe reescribirse utilizando vistas. Las técnicas para la reescritura de consultas difieren de manera importante según el enfoque de integración de bases de datos utilizado, es decir, GAV o LAV. En particular, las técnicas para LAV (y su extensión GLAV) son mucho más complejas. La mayor parte del trabajo en la reescritura de consultas utilizando vistas se ha realizado utilizando Datalog, que es un lenguaje de base de datos basado en la lógica. Datalog es más conciso que el cálculo relacional y, por lo tanto, más conveniente para describir algoritmos complejos de reescritura de consultas. En esta sección, primero presentamos la terminología de Datalog. Luego, describimos las principales técnicas y algoritmos para la reescritura de consultas en los enfoques GAV y LAV.

7.2.3.1 Terminología del registro de datos

Datalog puede considerarse una versión en línea del cálculo relacional de dominio. Definamos primero las consultas conjuntivas, es decir, las consultas de selección-proyecto-unión, que constituyen la base de consultas más complejas. Una consulta conjuntiva en Datalog se expresa como una regla de la forma:

$$Q(t) : -R_1(t_1), \dots, R_n(t_n)$$

El átomo $Q(t)$ es la cabecera de la consulta y denota la relación resultante. Los átomos $R_1(t_1), \dots$

, $R_n(t_n)$ son los subobjetivos en el cuerpo de la consulta y denotan relaciones de bases de datos. Q y R_1, \dots, R_n son nombres de predicados y corresponden a nombres de relaciones. t, t_1, \dots, t_n se refieren a las tuplas de relaciones y contienen variables o constantes. Las variables son similares a las variables de dominio en el cálculo relacional de dominio. Por lo tanto, el uso del mismo nombre de variable en varios predicados expresa predicados de unión equitativa.

Las constantes corresponden a predicados de igualdad. Los predicados de comparación más complejos (p. ej., que utilizan comparadores como $=, \leq$ y $<$) deben expresarse como otros subobjetivos.

Consideraremos consultas seguras, es decir, aquellas en las que cada variable del encabezado aparece también en el cuerpo. Las consultas disyuntivas también pueden expresarse en Datalog mediante uniones, al tener varias consultas conjuntivas con el mismo predicado principal.

Ejemplo 7.12 Consideremos las relaciones GCS EMP y WORKS definidas en la figura 7.9.

Considere la siguiente consulta SQL:

```
SELECCIONAR E#, TÍTULO, P#
DE EMP NATURAL JOIN WORKS
DONDE TÍTULO = "Programador" O DUR = 24
```

La consulta correspondiente en Datalog se puede expresar como:

```
Q(E#, TÍTULO, P#) : -EMP(E#, NOMBRE, "Programador", CIUDAD),
OBRAS(Mi#,P#,RESP,DUR)

Q(E#, TÍTULO, P#) : -EMP(E#, NOMBRE, TÍTULO, CIUDAD),
OBRAS(Mi#,P#,RESP,24)
```

7.2.3.2 Reescritura en GAV

En el enfoque GAV, el esquema global se expresa en términos de las fuentes de datos y cada relación global se define como una vista de las relaciones locales. Esto es similar a la definición del esquema global en los SGBD distribuidos y altamente integrados. En particular,

Las relaciones locales (es decir, las relaciones en un SGBD componente) pueden corresponder a fragmentos. Sin embargo, dado que las bases de datos locales preexisten y son autónomas, puede ocurrir que las tuplas de una relación global no existan en las relaciones locales, o que una tupla de una relación global aparezca en diferentes relaciones locales. Por lo tanto, no se pueden garantizar las propiedades de completitud y disyunción de la fragmentación. La falta de completitud puede generar respuestas incompletas a las consultas. La falta de disyunción puede generar resultados duplicados que, aun así, podrían ser información útil y no necesitar ser eliminada. Al igual que las consultas, las definiciones de vista pueden utilizar la notación Datalog.

Ejemplo 7.13 Consideremos las relaciones globales EMP y WORKS de la Fig. 7.9, con una ligera modificación: la responsabilidad predeterminada de un empleado en un proyecto corresponde a su cargo, por lo que el atributo TITLE está presente en la relación WORKS, pero ausente en la relación EMP. Consideremos las relaciones locales EMP1 y EMP2, cada una con los atributos E#, ENAME, TITLE y CITY, y la relación local WORKS1 con los atributos E#, P# y DUR. Las relaciones globales EMP y WORKS se pueden definir simplemente con las siguientes reglas de registro de datos:

$$\text{EMP}(\text{E}\#, \text{NOMBRE}, \text{CIUDAD}) : -\text{EMP1}(\text{E}\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}) \quad (\text{d1})$$

$$\text{EMP}(\text{E}\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}) : -\text{EMP2}(\text{E}\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}) \quad (\text{d2})$$

$$\begin{aligned} \text{OBRAS}(\text{E}\#, \text{P}\#, \text{TÍTULO}, \text{DUR}) : & -\text{EMP1}(\text{E}\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}), \\ & \text{OBRAS1}(\text{Mi}\#, \text{P}\#, \text{DUR}) \end{aligned} \quad (\text{d3})$$

$$\begin{aligned} \text{OBRAS}(\text{E}\#, \text{P}\#, \text{TÍTULO}, \text{DUR}) : & -\text{EMP2}(\text{E}\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD})), \\ & \text{OBRAS1}(\text{Mi}\#, \text{P}\#, \text{DUR}) \end{aligned} \quad (\text{d4})$$

Reescribir una consulta expresada en el esquema global en una consulta equivalente en las relaciones locales es relativamente sencillo y similar a la localización de datos en sistemas de gestión de bases de datos distribuidos altamente integrados (véase la sección 4.2). La técnica de reescritura que utiliza vistas se denomina desdoblamiento y reemplaza cada relación global invocada en la consulta por su vista correspondiente. Esto se logra aplicando las reglas de definición de vista a la consulta y generando una unión de consultas conjuntivas, una para cada aplicación de regla. Dado que una relación global puede definirse mediante varias reglas (véase el ejemplo 7.13), el desdoblamiento puede generar consultas redundantes que deben eliminarse.

Ejemplo 7.14 Consideremos el esquema global del Ejemplo 7.13 y la siguiente consulta q que solicita información de asignación sobre los empleados que viven en París:

$$Q(\text{e}, \text{p}) : -\text{EMP}(\text{e}, \text{ENAME}, \text{"París"}), \text{OBRAS}(\text{e}, \text{p}, \text{TÍTULO}, \text{DUR}).$$

Al desplegar q se obtiene q como sigue:

$$Q(e, p) : \neg \text{EMP1}(e, \text{ENAME}, \text{TÍTULO}, \text{"París"}, \text{OBRAS1}(e, p, \text{DUR})). \quad (\text{q1})$$

$$Q(e, p) : \neg \text{EMP2}(e, \text{ENAME}, \text{TÍTULO}, \text{"París"}, \text{OBRAS1}(e, p, \text{DUR})). \quad (\text{q2})$$

Q es la unión de dos consultas conjuntivas denominadas q1 y q2. q1 se obtiene aplicando la regla d3 de GAV o ambas reglas . En este último caso, la consulta obtenida es redundante con respecto a la obtenida solo con d3 . De forma similar, q2 se obtiene aplicando la regla d4 o ambas reglas .

Aunque la técnica básica es sencilla, la reescritura en GAV se dificulta cuando las bases de datos locales tienen patrones de acceso limitados. Este es el caso de las bases de datos a las que se accede a través de la web, donde solo se puede acceder a las relaciones mediante ciertos patrones de enlace para sus atributos. En este caso, simplemente sustituir las relaciones globales por sus vistas no es suficiente, y la reescritura de consultas requiere el uso de consultas recursivas de registro de datos.

7.2.3.3 Reescritura en LAV

En el enfoque LAV, el esquema global se expresa independientemente de las bases de datos locales y cada relación local se define como una vista de las relaciones globales. Esto permite una considerable flexibilidad para definir las relaciones locales.

Ejemplo 7.15 Para facilitar la comparación con GAV, desarrollamos un ejemplo simétrico al Ejemplo 7.13 , donde EMP y WORKS se definen como relaciones globales. En el enfoque LAV, las relaciones locales EMP1, EMP2 y WORKS1 se pueden definir con las siguientes reglas de registro de datos:

$$\begin{aligned} \text{EMP1}(E\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}) : & \neg \text{EMP}(E\#, \text{NOMBRE}, \text{CIUDAD}), \\ & \text{OBRAS}(M\#, P\#, \text{TÍTULO}, \text{DURACIÓN}) \end{aligned} \quad (\text{d5})$$

$$\begin{aligned} \text{EMP2}(E\#, \text{NOMBRE}, \text{TÍTULO}, \text{CIUDAD}) : & \neg \text{EMP}(E\#, \text{NOMBRE}, \text{CIUDAD}), \\ & \text{OBRAS}(M\#, P\#, \text{TÍTULO}, \text{DURACIÓN}) \end{aligned} \quad (\text{d6})$$

$$\text{OBRAS1}(E\#, P\#, \text{DUR}) : \neg \text{OBRAS}(E\#, P\#, \text{TÍTULO}, \text{DUR}) \quad (\text{d7})$$

Reescribir una consulta expresada en el esquema global en una consulta equivalente en las vistas que describen las relaciones locales es difícil por tres razones. En primer lugar, a diferencia del enfoque GAV, no existe una correspondencia directa entre los términos utilizados en el esquema global (p. ej., EMP, ENAME) y los utilizados en las vistas (p. ej., EMP1, EMP2, ENAME). Encontrar las correspondencias requiere comparar cada vista. En segundo lugar, puede haber muchas más vistas que relaciones globales, lo que hace que la comparación de vistas sea lenta. En tercer lugar, las definiciones de vista pueden contener predicados complejos para reflejar el contenido específico de las relaciones locales (p. ej., vista EMP3).

Contiene solo programadores. Por lo tanto, no siempre es posible encontrar una reescritura equivalente de la consulta. En este caso, lo mejor es encontrar una consulta con el máximo contenido , es decir, una que produzca el máximo subconjunto de la respuesta.

Por ejemplo, EMP3 sólo podría devolver un subconjunto de todos los empleados, aquellos que son programadores.

La reescritura de consultas mediante vistas ha recibido mucha atención debido a su relevancia para problemas de integración de datos, tanto lógicos como físicos. En el contexto de la integración física (es decir, el almacenamiento de datos), el uso de vistas materializadas puede ser mucho más eficiente que el acceso a relaciones base. Sin embargo, el problema de encontrar una reescritura mediante vistas es NP-completo en cuanto al número de vistas y subobjetivos de la consulta. Por lo tanto, los algoritmos para reescribir una consulta mediante vistas intentan, esencialmente, reducir el número de reescrituras necesarias. Se han propuesto tres algoritmos principales para este propósito: el algoritmo de cubo, el algoritmo de regla inversa y el algoritmo MinCon. El algoritmo de cubo y el algoritmo de regla inversa presentan limitaciones similares, que son abordadas por el algoritmo MinCon.

El algoritmo de cubo considera cada predicado de la consulta de forma independiente para seleccionar solo las vistas relevantes para dicho predicado. Dada una consulta Q, el algoritmo se desarrolla en dos pasos. En el primero, crea un cubo b para cada subobjetivo q de Q que no sea un predicado de comparación e inserta en b los encabezados de las vistas relevantes para la respuesta q. Para determinar si una vista V debe estar en b, debe existir una asignación que unifique q con un subobjetivo v en V.

Por ejemplo, considere la consulta Q en el Ejemplo 7.14 y las vistas en el Ejemplo 7.15.

La siguiente asignación unifica el subobjetivo EMP(e, ENAME, "París") de Q con el subobjetivo EMP(E#, ENAME, CITY) en la vista EMP1:

$$e \rightarrow E\#, \text{"París"} \rightarrow \text{CIUDAD}$$

En el segundo paso, para cada vista V del producto cartesiano de los cubos no vacíos (es decir, algún subconjunto de los cubos), el algoritmo produce una consulta conjuntiva y verifica si está contenido en Q. Si lo está, la consulta conjuntiva se mantiene ya que representa una forma de responder parte de Q desde V. Por lo tanto, la consulta reescrita es una unión de consultas conjuntivas.

Ejemplo 7.16 Consideremos la consulta Q del Ejemplo 7.14 y las vistas del Ejemplo 7.15. En el primer paso, el algoritmo de cubos crea dos cubos, uno para cada subobjetivo de Q. Denotemos con b1 el cubo para el subobjetivo EMP(e, ENAME, "París") y con b2 el cubo para el subobjetivo WORKS(e, p, TITLE, DUR). Dado que el algoritmo solo inserta los encabezados de vista en un cubo, puede haber variables en un encabezado de vista que no estén en la asignación unificadora. Dichas variables simplemente se priman.

Obtenemos los siguientes cubos:

$$\begin{aligned} b1 = & \{ \text{EMP1}(E\#, ENAME, TÍTULO}, & , \text{CIUDAD}), \\ & \text{EMP2}(E\#, ENAME, TÍTULO}, & , \text{CIUDAD}) \} \\ b2 = & \{ \text{TRABAJOS1}(E\#, P\#, DUR) \} \end{aligned}$$

En el segundo paso, el algoritmo combina los elementos de los contenedores, lo que produce una unión de dos consultas conjuntivas:

$$Q(e, p) : \neg \text{EMP1}(e, \text{ENAME}, \text{TÍTULO}, \text{"París"}, \text{OBRAS1}(e, p, \text{DUR})) \quad (q1)$$

$$Q(e, p) : \neg \text{EMP2}(e, \text{ENAME}, \text{TÍTULO}, \text{"París"}, \text{OBRAS1}(e, p, \text{DUR})) \quad (q2)$$

La principal ventaja del algoritmo de cubos es que, al considerar los predicados de la consulta, se puede reducir significativamente el número de reescrituras necesarias. Sin embargo, considerar los predicados de la consulta de forma aislada puede resultar en la adición de una vista a un cubo que resulta irrelevante al considerar la unión con otras vistas. Además, el segundo paso del algoritmo puede generar un gran número de reescrituras como resultado del producto cartesiano de los cubos.

Ejemplo 7.17 Consideremos la consulta Q del Ejemplo 7.14 y las vistas del Ejemplo 7.15 con la adición de la siguiente vista que proporciona los proyectos para los cuales hay empleados que viven en París.

$$\begin{aligned} \text{PROJ1}(P\#) : & \neg \text{EMP1}(E\#, \text{ENAME}, \text{"París"}), \\ & \text{OBRAS}(Mi\#, P\#, \text{TÍTULO}, \text{DURACIÓN}) \end{aligned} \quad (d8)$$

Ahora, la siguiente asignación unifica el subobjetivo WORKS(e, p, TITLE, DUR) de Q con el subobjetivo WORKS(E#, P#, TITLE, DUR) en la vista PROJ1:

$$p \rightarrow \text{PNAME}$$

Por lo tanto, en el primer paso del algoritmo del depósito, PROJ1 se agrega al depósito b2.

Sin embargo, PROJ1 no puede ser útil en una reescritura de Q ya que la variable ENAME no está en la cabecera de PROJ1 y por lo tanto hace imposible unir PROJ1 en la variable e de Q. Esto se puede descubrir solo en el segundo paso al construir las consultas conjuntivas.

El algoritmo MinCon aborda las limitaciones del algoritmo de cubo (y del algoritmo de regla inversa) considerando la consulta globalmente y cómo cada predicado de la consulta interactúa con las vistas. Al igual que el algoritmo de cubo, se desarrolla en dos pasos. El primer paso comienza seleccionando las vistas que contienen subobjetivos correspondientes a los subobjetivos de la consulta Q. Sin embargo, al encontrar una asignación que unifique un subobjetivo q de Q con un subobjetivo v en la vista V, considera los predicados de unión en Q y encuentra el conjunto mínimo de subobjetivos adicionales de Q que deben asignarse a los subobjetivos de V. Este conjunto de subobjetivos de Q se captura mediante una descripción MinCon (MCD) asociada a V. El segundo paso del algoritmo produce una consulta reescrita combinando las diferentes MCD. En este segundo paso, a diferencia del algoritmo de cubo, no es necesario comprobar que las reescrituras propuestas estén contenidas en la consulta.

porque la forma en que se crean los MCD garantiza que las reescrituras resultantes estarán contenidas en la consulta original.

Aplicado al Ejemplo 7.17, el algoritmo crearía tres MCD: dos para las vistas EMP1 y EMP2 que contienen el subobjetivo EMP de Q, y uno para ASG1 que contiene el subobjetivo ASG. Sin embargo, el algoritmo no puede crear un MCD para PROJ1 porque no puede aplicar el predicado de unión en Q. Por lo tanto, el algoritmo generaría la consulta reescrita Q del Ejemplo 7.16. En comparación con el algoritmo de cubos, el segundo paso del algoritmo MinCon es mucho más eficiente, ya que realiza menos combinaciones de MCD que de cubos.

7.2.4 Optimización y ejecución de consultas

Los tres problemas principales de la optimización de consultas en sistemas multibase de datos son el modelado de costes heterogéneo, la optimización de consultas heterogénea (para gestionar las diferentes capacidades de los SGBD que los componen) y el procesamiento adaptativo de consultas (para gestionar las fuertes variaciones del entorno: fallos, retrasos impredecibles, etc.). En esta sección, describimos las técnicas para los dos primeros problemas. En la sección 4.6, presentamos las técnicas para el procesamiento adaptativo de consultas. Estas técnicas también pueden utilizarse en sistemas multibase de datos, siempre que los wrappers puedan recopilar información sobre la ejecución dentro de los SGBD que los componen.

7.2.4.1 Modelado de costos heterogéneos

La definición de la función de coste global y el problema asociado de obtener información sobre costes de los SGBD que la componen es quizás el más estudiado de los tres problemas. Han surgido diversas soluciones posibles, que se analizan a continuación.

Lo primero que hay que tener en cuenta es que nos interesa principalmente determinar el coste de los niveles inferiores de un trie de ejecución de consultas, que corresponden a las partes de la consulta ejecutadas en los SGBD de los componentes. Si asumimos que todo el procesamiento local se traslada al trie, podemos modificar el plan de consulta para que las hojas del trie correspondan a subconsultas que se ejecutarán en los SGBD de los componentes individuales. En este caso, nos referimos a la determinación de los costes de estas subconsultas que se introducen en los operadores de primer nivel (desde abajo). El coste de los niveles superiores del trie de ejecución de consultas puede calcularse recursivamente, basándose en los costes de los nodos hoja.

Existen tres enfoques alternativos para determinar el costo de ejecutar consultas en los DBMS componentes:

1. Enfoque de caja negra. Este enfoque trata cada componente del SGBD como una caja negra, ejecutando consultas de prueba en él y, a partir de ellas, determina la información de costos necesaria.

2. Enfoque personalizado. Este enfoque utiliza el conocimiento previo sobre los componentes del SGBD, así como sus características externas, para determinar subjetivamente la información de costos.

3. Enfoque dinámico. Este enfoque monitorea el comportamiento en tiempo de ejecución de los componentes DBMS existentes y recopila dinámicamente la información de costos.

Analizamos cada enfoque, centrándonos en las propuestas que han atraído más atención.

Enfoque de caja negra

En el enfoque de caja negra, las funciones de costo se expresan lógicamente (p. ej., costos agregados de CPU y E/S, factores de selectividad), en lugar de basarse en características físicas (p. ej., cardinalidades de relación, número de páginas, número de valores distintos para cada columna). Por lo tanto, las funciones de costo de los SGBD de componentes se expresan como

$$\text{Costo} = \text{costo de inicialización} + \text{costo para encontrar tuplas calificadas}$$

$$+ \text{costo de procesar tuplas seleccionadas}$$

Los términos individuales de esta fórmula varíanán según el operador. Sin embargo, estas diferencias no son difíciles de especificar a priori. La dificultad fundamental reside en la determinación de los coeficientes de los términos en estas fórmulas, que varían según los diferentes SGBD que los componen. Una forma de solucionar esto es construir una base de datos sintética (denominada base de datos de calibración), ejecutar consultas en ella de forma aislada y medir el tiempo transcurrido para deducir los coeficientes.

Un problema con este enfoque es que los resultados obtenidos con una base de datos sintética podrían no ser aplicables a sistemas de gestión de bases de datos reales. Una alternativa consiste en ejecutar consultas de sondeo en los sistemas de gestión de bases de datos de componentes para determinar la información de costes. De hecho, estas consultas pueden utilizarse para recopilar diversos factores de información de costes. Por ejemplo, se pueden ejecutar consultas de sondeo para recuperar datos de los sistemas de gestión de bases de datos de componentes y así construir y actualizar el catálogo multibase de datos. Se pueden ejecutar consultas de sondeo estadísticas que, por ejemplo, cuenten el número de tuplas de una relación. Finalmente, se pueden ejecutar consultas de sondeo para medir el rendimiento y medir el tiempo transcurrido para determinar los coeficientes de la función de costes.

Un caso especial de consultas de sondeo son las consultas de muestra. En este caso, las consultas se clasifican según diversos criterios, y se emiten y miden consultas de muestra de cada clase para obtener información sobre el coste de los componentes. La clasificación de las consultas puede realizarse según sus características (p. ej., consultas de operación unaria, consultas de unión bidireccional), las características de las relaciones entre operandos (p. ej., cardinalidad, número de atributos, información sobre atributos indexados) y las características de los SGBD de los componentes subyacentes (p. ej., los métodos de acceso admitidos y las políticas para su selección).

Las reglas de clasificación se definen para identificar consultas que se ejecutan de forma similar y, por lo tanto, podrían compartir la misma fórmula de coste. Por ejemplo, se puede considerar que dos consultas con expresiones algebraicas similares (es decir, la misma forma de trío algebraico), pero diferentes relaciones de operandos, atributos o constantes, se ejecutan de la misma manera si sus atributos tienen las mismas propiedades físicas. Otro ejemplo es asumir que el orden de unión de una consulta no afecta a la ejecución, ya que el optimizador de consultas subyacente aplica técnicas de reordenación para elegir un orden de unión eficiente. Por lo tanto, dos consultas que unen el mismo conjunto de relaciones pertenecen a la misma clase, independientemente del orden expresado por el usuario. Las reglas de clasificación se combinan para definir las clases de consulta. La clasificación se realiza de arriba a abajo, dividiendo una clase en clases más específicas, o de abajo a arriba, fusionando dos clases en una más grande. En la práctica, se obtiene una clasificación eficiente combinando ambos enfoques. La función de coste global consta de tres componentes: coste de inicialización, coste de recuperación de una tupla y coste de procesamiento de una tupla. La diferencia radica en cómo se determinan los parámetros de esta función. En lugar de usar una base de datos de calibración, se ejecutan consultas de muestra y se miden los costos. La ecuación de costo global se trata como una ecuación de regresión, y los coeficientes de regresión se calculan utilizando los costos medidos de las consultas de muestra. Los coeficientes de regresión son los parámetros de la función de costo. Finalmente, la calidad del modelo de costos se controla mediante pruebas estadísticas (p. ej., la prueba F): si las pruebas fallan, la clasificación de las consultas se refina hasta que la calidad sea suficiente.

Los enfoques anteriores requieren un paso preliminar para instanciar el modelo de costos (ya sea mediante calibración o muestreo). Esto puede no ser siempre apropiado, ya que ralentizaría el sistema cada vez que se añade un nuevo componente del SGBD. Una forma de solucionar este problema es aprender progresivamente el modelo de costos a partir de consultas. Se asume que el mediador invoca los componentes subyacentes del DBMS mediante una llamada a una función. El coste de una llamada se compone de tres valores: el tiempo de respuesta para acceder a la primera tupla, el tiempo de respuesta del resultado completo y la cardinalidad del resultado. Esto permite al optimizador de consultas minimizar el tiempo para recibir la primera tupla o el tiempo para procesar la consulta completa, según los requisitos del usuario final. Inicialmente, el procesador de consultas desconoce las estadísticas sobre los componentes del DBMS. Posteriormente, monitoriza las consultas en curso: recopila el tiempo de procesamiento de cada llamada y lo almacena para futuras estimaciones. Para gestionar la gran cantidad de estadísticas recopiladas, el gestor de costes las resume, ya sea sin pérdida de precisión o con menor precisión, con el beneficio de un menor uso de espacio y una estimación de costes más rápida. La suma consiste en agregar estadísticas: se calcula el tiempo de respuesta medio para todas las llamadas que coinciden con el mismo patrón, es decir, aquellas con el mismo nombre de función y cero o más valores de argumento idénticos. El módulo estimador de costes se implementa en un lenguaje declarativo. Esto permite añadir nuevas fórmulas de coste que describen el comportamiento de un componente específico del DBMS. Sin embargo, la carga de ampliar el modelo de costos del mediador recae en el desarrollador del mediador.

La principal desventaja del enfoque de caja negra es que el modelo de costos, aunque ajustado mediante calibración, es común para todos los SGBD que lo componen y podría no reflejar sus particularidades. Por lo tanto, podría no estimar con precisión el costo de una consulta ejecutada en un SGBD que exponga un comportamiento imprevisto.

Enfoque personalizado

Este enfoque se basa en que los procesadores de consultas de los SGBD componentes son demasiado diferentes como para ser representados por un modelo de costos único, como el utilizado en el enfoque de caja negra. También asume que la capacidad de estimar con precisión el costo de las subconsultas locales mejorará la optimización global de consultas. Este enfoque proporciona un marco para integrar el modelo de costos de los SGBD componentes en el optimizador de consultas del mediador. La solución consiste en extender la interfaz del contenedor de forma que el mediador obtenga información específica de costos de cada contenedor. El desarrollador del contenedor puede proporcionar un modelo de costos, parcial o totalmente. El reto consiste entonces en integrar esta descripción de costos (potencialmente parcial) en el optimizador de consultas del mediador. Existen dos soluciones principales.

Una primera solución consiste en proporcionar la lógica dentro del contenedor para calcular tres estimaciones de costos: el tiempo para iniciar el proceso de consulta y recibir el primer resultado (denominado `reset_cost`), el tiempo para obtener el siguiente resultado (denominado `advance_cost`) y la cardinalidad del resultado. Por lo tanto, el costo total de la consulta es

$$\text{Costo_de_acceso_total} = \text{costo_de_reinicio} + (\text{cardinalidad} - 1) \cdot \text{costo_de_avance}$$

Esta solución puede extenderse para estimar el coste de las llamadas a procedimientos de bases de datos. En ese caso, el contenedor proporciona una fórmula de coste que es una ecuación lineal que depende de los parámetros del procedimiento. Esta solución se ha implementado con éxito para modelar una amplia gama de sistemas de gestión de bases de datos (SGBD) con componentes heterogéneos, desde un SGBD relacional hasta un servidor de imágenes. Esto demuestra que basta con un pequeño esfuerzo para implementar un modelo de coste bastante simple, lo que mejora significativamente el procesamiento distribuido de consultas en fuentes heterogéneas.

Una segunda solución consiste en utilizar un modelo de costes genérico jerárquico. Como se muestra en la figura 7.14, cada nodo representa una regla de costes que asocia un patrón de consulta con una función de costes para diversos parámetros de coste.

La jerarquía de nodos se divide en cinco niveles según la genericidad de las reglas de coste (en la Fig. 7.14, el aumento del ancho de los cuadros muestra el mayor enfoque de las reglas). En el nivel superior, las reglas de coste se aplican por defecto a cualquier SGBD. En los niveles subyacentes, las reglas de coste se centran cada vez más en: SGBD, relaciones, predicados o consultas específicos. Al registrar el contenedor, el mediador recibe sus metadatos, incluyendo información de costes, y completa su modelo de costes integrado añadiendo nuevos nodos en el nivel correspondiente de la jerarquía.

Este marco es lo suficientemente general como para capturar e integrar tanto el conocimiento general de costos declarado como reglas proporcionadas por los desarrolladores de wrappers como la información específica derivada de consultas anteriores registradas. Por lo tanto, mediante una jerarquía de herencia, el optimizador basado en costos del mediador puede admitir una amplia variedad de fuentes de datos. El mediador se beneficia de información especializada sobre los costos de cada componente del SGBD para estimar con precisión el costo de las consultas y elegir un QEP más eficiente.

Ejemplo 7.18. Considere las relaciones GCS EMP y WORKS (Fig. 7.9). EMP se almacena en el componente DBMS db1 y contiene 1000 tuplas. ASG se almacena en el componente DBMS db2 y contiene 10 000 tuplas. Suponemos una distribución uniforme.

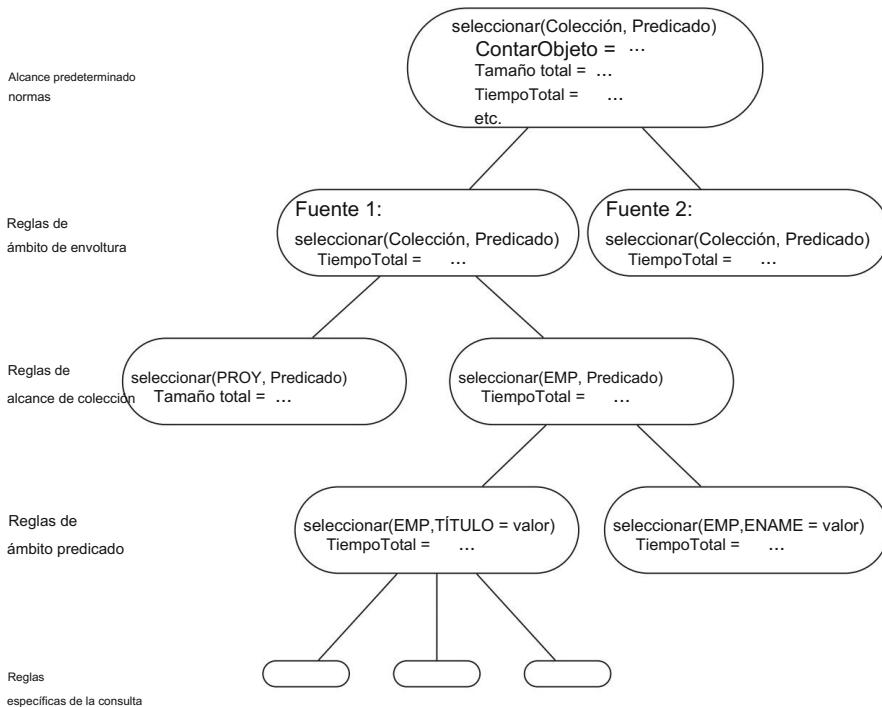


Fig. 7.14 Fórmula de costo jerárquico trie

de valores de atributos. La mitad de las tuplas WORKS tienen una duración mayor a 6. A continuación se detallan algunas partes del modelo de costos genérico del mediador donde R y S son dos relaciones, A es el atributo de unión y usamos superíndices para indicar el acceso método:

$$\text{costo (R)} = |R|$$

costo ($\sigma_{\text{predicado}(R)}$) = costo (R) (acceso a R mediante escaneo secuencial, por defecto)

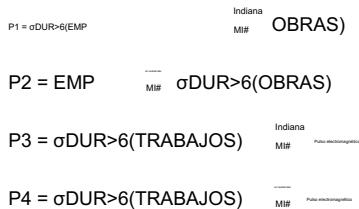
$$\text{costo (R} \underset{\substack{\text{Indiana} \\ A}}{\text{---}} \text{S)} = \text{costo (R)} + |R| \quad \text{costo } (\sigma_{A=v(S)}) \text{ (usando un índice basado en (ind) se une con el índice en SA)}$$

$$\text{costo (R} \underset{\substack{\text{---} \\ A}}{\text{---}} \text{S)} = \text{costo (R)} + |R| \quad \text{costo (S)} \text{ (usando una unión de bucle anidado (nl))}$$

Considere la siguiente consulta global Q:

```
SELECCIONAR *
DE EMP NATURAL JOIN WORKS
DONDE TRABAJA.DUR>6
```

El optimizador de consultas basado en costos genera los siguientes planes para procesar Q:



Basándonos en el modelo de costes genérico, calculamos su coste como:

$$\begin{aligned}
 \text{costo (P1)} &= \text{costo } (\sigma DUR > 6(\text{EMP})) \quad \text{Indiana} \\
 &\quad \text{Mi#} \quad \text{OBRAS}) \\
 &= \text{costo } (\text{EMP} \quad \text{Indiana} \quad \text{Mi#} \quad \text{OBRAS}) \\
 &= \text{costo } (\text{EMP}) + |\text{EMP}| \quad \text{costo } (\sigma E\#=v(\text{TRABAJOS})) \\
 &= |\text{EMP}| + |\text{EMP}| \quad |\text{FUNCIONA}| = 10.001.000 \\
 \text{costo (P2)} &= \text{costo } (\text{EMP}) + |\text{EMP}| \quad \text{costo } (\sigma DUR > 6(\text{WORKS})) \\
 &= \text{costo } (\text{EMP}) + |\text{EMP}| \quad \text{costo } (\text{OBRAS}) \\
 &= |\text{EMP}| + |\text{EMP}| \quad |\text{FUNCIONA}| = 10.001.000 \\
 \text{costo (P3)} &= \text{costo (P4)} = |\text{OBRAS}| + |\text{EMP}| \frac{|\text{OBRAS}|}{2} \\
 &= 5.010.000
 \end{aligned}$$

Por lo tanto, el optimizador descarta los planes P1 y P2 para conservar P3 o P4 para el procesamiento. P. Supongamos ahora que el mediador importa información de costos específica sobre componentes DBMS. db1 exporta el costo de acceso a las tuplas EMP como:

$$\text{costo } (\sigma A=v(R)) = |\sigma A=v(R)|$$

db2 exporta el costo específico de seleccionar tuplas WORKS que tienen un E# determinado como:

$$\text{costo } (\sigma E\#=v(\text{TRABAJO})) = |\sigma E\#=v(\text{TRABAJO})|$$

El mediador integra estas funciones de costos en su modelo de costos jerárquico y puede Ahora estimamos con mayor precisión el coste de los QEP:

$$\text{costo (P1)} = |\text{EMP}| + |\text{EMP}| \quad |\sigma E\#=v(\text{TRABAJO})|$$

$$= 1.000 + 1.000 \quad 10$$

$$= 11.000$$

$$\text{costo (P2)} = |\text{EMP}| + |\text{EMP}| \cdot |\sigma_{\text{DUR} > 6}(\text{WORKS})|$$

$$= |\text{EMP}| + |\text{EMP}| \cdot \frac{|\text{ASG}|}{2}$$

$$= 5.001.000$$

$$\text{costo (P3)} = |\text{OBRAS}| + |\text{oE} \# \neq v(\text{EMP})| \cdot \frac{|\text{OBRAS}|}{2}$$

$$= 10.000 + 5.000 \cdot 1$$

$$= 15.000$$

$$\text{costo (P4)} = |\text{OBRAS}| + |\text{EMP}| \cdot \frac{|\text{OBRAS}|}{2}$$

$$= 10.000 + 5.000 \cdot 1.000$$

$$= 5.010.000$$

El mejor QEP ahora es P1 , que anteriormente se descartó por falta de información sobre el costo de los SGBD que lo componen. En muchas situaciones, P1 es la mejor alternativa al proceso Q1.

Las dos soluciones que acabamos de presentar se adaptan bien a la arquitectura de mediador/ envoltorio y ofrecen un buen equilibrio entre la sobrecarga que supone proporcionar información de costes específica para los diversos componentes de los DBMS y el beneficio de un procesamiento de consultas heterogéneas más rápido.

Enfoque dinámico

Los enfoques anteriores suponen que el entorno de ejecución es estable a lo largo del tiempo. Sin embargo, en la mayoría de los casos, los factores del entorno de ejecución cambian con frecuencia. Se pueden identificar tres clases de factores ambientales en función de su dinamismo. La primera clase, para factores que cambian con frecuencia (cada segundo o cada minuto), incluye la carga de la CPU, el rendimiento de E/S y la memoria disponible. La segunda clase, para factores que cambian lentamente (cada hora o cada día), incluye los parámetros de configuración del SGBD, la organización física de los datos en discos y el esquema de la base de datos. La tercera clase, para factores casi estables (cada mes o cada año), incluye el tipo de SGBD, la ubicación de la base de datos y la velocidad de la CPU. Nos centramos en soluciones que abordan las dos primeras clases.

Una forma de abordar entornos dinámicos donde la contención de la red, el almacenamiento de datos o la memoria disponible cambian con el tiempo es ampliar el método de muestreo y considerar las consultas de los usuarios como nuevas muestras. El tiempo de respuesta a las consultas se mide.

para ajustar los parámetros del modelo de costos en tiempo de ejecución para consultas posteriores. Esto evita la sobrecarga de procesar consultas de muestra periódicamente, pero aún requiere cálculos pesados para resolver las ecuaciones del modelo de costos y no garantiza que la precisión del modelo de costos mejore con el tiempo. Una mejor solución, llamada cualitativa, define el nivel de contención del sistema como el efecto combinado de factores que cambian frecuentemente en el costo de la consulta. El nivel de contención del sistema se divide en varias categorías discretas: alta, media, baja o sin contención del sistema. Esto permite definir un modelo de costos multicategoría que proporciona estimaciones de costos precisas, mientras que los factores dinámicos varían. El modelo de costos se calibra inicialmente utilizando consultas de sondeo. El nivel de contención del sistema actual se calcula con el tiempo, con base en los parámetros del sistema más significativos. Este enfoque asume que las ejecuciones de consultas son cortas, por lo que los factores del entorno permanecen bastante constantes durante la ejecución de la consulta. Sin embargo, esta solución no se aplica a consultas de ejecución larga, ya que los factores del entorno pueden cambiar rápidamente durante la ejecución de la consulta.

Para gestionar los casos en que la variación del factor de entorno es predecible (por ejemplo, la variación diaria de la carga del DBMS es la misma todos los días), se calcula el coste de la consulta para rangos de fechas sucesivos. Posteriormente, el coste total es la suma de los costes de cada rango. Además, es posible conocer el patrón del ancho de banda de red disponible entre el procesador de consultas MDBS y el DBMS del componente. Esto permite ajustar el coste de la consulta en función de la fecha real.

7.2.4.2 Optimización de consultas heterogéneas

Además del modelado de costos heterogéneos, la optimización de consultas multibase de datos debe abordar el problema de las capacidades computacionales heterogéneas de los SGBD que los componen. Por ejemplo, un SGBD que los compone puede admitir solo operaciones de selección simples, mientras que otro puede admitir consultas complejas que involucran uniones y agregaciones. Por lo tanto, dependiendo de cómo los wrappers exporten dichas capacidades, el procesamiento de consultas a nivel de mediador puede ser más o menos complejo. Existen dos enfoques principales para abordar este problema, según el tipo de interfaz entre el mediador y el wrapper: basado en consultas y basado en operadores.

1. Basado en consultas. En este enfoque, los envoltorios admiten la misma capacidad de consulta (p. ej., un subconjunto de SQL), que se traduce a la capacidad del SGBD componente. Este enfoque suele basarse en una interfaz estándar de SGBD, como Conectividad Abierta de Bases de Datos (ODBC) y sus extensiones para los envoltorios, o en la Gestión SQL de Datos Externos (SQL/MED). Por lo tanto, dado que los SGBD componentes parecen homogéneos al mediador, se pueden reutilizar las técnicas de procesamiento de consultas diseñadas para SGBD distribuidos homogéneos. Sin embargo, si los SGBD componentes tienen capacidades limitadas, las capacidades adicionales deben implementarse en los envoltorios; por ejemplo, las consultas de unión podrían tener que gestionarse en el envoltorio si el SGBD componente no admite la unión.
2. Basado en operadores. En este enfoque, los contenidos exportan las capacidades de los SGBD componentes mediante composiciones de operadores relacionales. Por lo tanto, existe

Mayor flexibilidad para definir el nivel de funcionalidad entre el mediador y el contenedor. En particular, las diferentes capacidades de los SGBD componentes pueden ponerse a disposición del mediador. Esto facilita la construcción del contenedor, a costa de un procesamiento de consultas más complejo en el mediador. En particular, cualquier funcionalidad que no sea compatible con los SGBD componentes (p. ej., la unión) deberá implementarse en el mediador.

En el resto de esta sección, presentamos con más detalle los enfoques para la optimización de consultas en estos sistemas.

Enfoque basado en consultas

Dado que los SGBD componentes parecen homogéneos para el mediador, un enfoque consiste en utilizar un algoritmo de optimización de consultas basado en costes distribuidos (véase el cap. 4) con un modelo de costes heterogéneo (véase la sec. 7.2.4.1). Sin embargo, se requieren extensiones para convertir el plan de ejecución distribuido en subconsultas que ejecutarán los SGBD componentes y en subconsultas que ejecutará el mediador. La técnica de optimización híbrida en dos pasos resulta útil en este caso (véase la sec. 4.5.3): en el primer paso, un optimizador de consultas basado en costes centralizado genera un plan estático; en el segundo paso, al iniciar el sistema, se genera un plan de ejecución seleccionando el sitio y asignando las subconsultas a los sitios. Sin embargo, los optimizadores centralizados restringen su espacio de búsqueda eliminando la consideración de árboles de unión frondosos.

Casi todos los sistemas utilizan órdenes de unión lineal izquierda. Considerar únicamente árboles de unión lineal izquierda ofrece buenos resultados en sistemas de gestión de bases de datos centralizados por dos razones: reduce la necesidad de estimar estadísticas para al menos un operando y permite explotar los índices de uno de ellos. Sin embargo, en sistemas multibase de datos, estos planes de ejecución de unión no son necesariamente los preferidos, ya que no permiten el paralelismo en la ejecución. Como se explicó en capítulos anteriores, esto también representa un problema en sistemas de gestión de bases de datos distribuidos homogéneos, pero el problema es más grave en el caso de sistemas multibase de datos, ya que se busca transferir el máximo procesamiento posible a los sistemas que los componen.

Una forma de resolver este problema es generar de alguna manera árboles de unión arbustivos y considerarlos a expensas de los lineales izquierdos. Una forma de lograr esto es aplicar un optimizador de consultas basado en costos para generar primero un trie de unión lineal izquierda y luego convertirlo en un trie arbustivo. En este caso, el plan de ejecución de la unión lineal izquierda puede ser óptimo con respecto al tiempo total, y la transformación mejora el tiempo de respuesta de la consulta sin afectar gravemente el tiempo total. Es posible un algoritmo híbrido que realiza simultáneamente un barrido de abajo a arriba y de arriba a abajo del trie de ejecución de la unión lineal izquierda, transformándolo, paso a paso, en uno arbustivo. El algoritmo mantiene dos punteros, llamados nodos de anclaje superiores (UAN) en el trie. Al principio, uno de estos, llamado UAN inferior (UANB), se establece en el abuelo del nodo raíz más a la izquierda (unión con R3 en la Fig. 4.9), mientras que el segundo, llamado UAN superior (UANT), se establece en la raíz (unión con R5). Para cada UAN, el algoritmo selecciona un nodo de anclaje inferior (LAN). Este es el nodo más cercano a la UAN y cuyo lado derecho

El tiempo de respuesta del subárbol hijo se encuentra dentro de un rango especificado por el diseñador, en relación con el del subárbol hijo derecho de la UAN. Intuitivamente, la LAN se elige de forma que el tiempo de respuesta de su subárbol hijo derecho se aproxime al del subárbol hijo derecho de la UAN correspondiente. Como veremos en breve, esto ayuda a mantener equilibrado el trie arbustivo transformado, lo que reduce el tiempo de respuesta.

En cada paso, el algoritmo elige uno de los pares UAN/LAN (estrictamente hablando, elige la UAN y selecciona la LAN apropiada, como se explicó anteriormente) y realiza la siguiente traducción para el segmento entre ese par LAN y UAN:

1. El hijo izquierdo de UAN se convierte en el nuevo UAN del segmento transformado.
2. La LAN permanece sin cambios, pero su vértice secundario derecho se reemplaza con un nuevo nodo de unión de dos subárboles, que eran los subárboles secundarios derechos de la UAN y LAN de entrada.

El modo UAN que se considerará en esa iteración particular se elige de acuerdo con la siguiente heurística: elija UANB si el tiempo de respuesta de su subárbol hijo izquierdo es menor que el del subárbol de UANT; de lo contrario, elija UANT . Si

los tiempos de respuesta son los mismos, elija el que tenga el subárbol secundario más desequilibrado.

Al final de cada paso de transformación, se ajustan la UANB y la UANT .

El algoritmo finaliza cuando UANB = UANT , ya que esto indica que no son posibles más transformaciones. El trie de ejecución de la unión resultante estará prácticamente equilibrado, lo que generará un plan de ejecución cuyo tiempo de respuesta se reduce debido a la ejecución paralela de las uniones.

El algoritmo descrito anteriormente comienza con un trie de ejecución de unión lineal izquierda generado por un optimizador de SGBD centralizado. Estos optimizadores pueden generar planes muy buenos, pero el plan de ejecución lineal inicial podría no considerar completamente las peculiaridades de las características de las multibases de datos distribuidas, como la replicación de datos. Un algoritmo especial de optimización de consultas globales puede tener esto en cuenta. Un algoritmo propuesto parte de un plan inicial y verifica diferentes paréntesis en el orden de ejecución de esta unión lineal para producir un orden entre paréntesis óptimo en cuanto al tiempo de respuesta. El resultado es un trie de ejecución de unión (casi) equilibrado. Es probable que este enfoque produzca planes de mejor calidad a costa de un mayor tiempo de optimización.

Enfoque basado en el operador

Expresar las capacidades de los componentes DBMS a través de operadores relationales permite una integración estrecha del procesamiento de consultas entre el mediador y los envoltorios.

En particular, la comunicación mediador/contenedor puede ser en términos de subplanes.

Ilustramos el enfoque basado en operadores mediante las funciones de planificación propuestas en el proyecto Garlic. En este enfoque, las capacidades de los SGBD componentes se expresan mediante los envoltorios como funciones de planificación que pueden ser invocadas directamente por un optimizador de consultas centralizado. Extiende un optimizador basado en reglas con operadores para crear relaciones temporales y recuperar datos almacenados localmente. También crea...

Operador PushDown que envía una parte del trabajo a los SGBD de los componentes, donde se ejecutará. Los planes de ejecución se representan, como es habitual, como árboles de operadores, pero los nodos de operador se anotan con información adicional que especifica el origen de los operandos, si los resultados se materializan, etc. Los árboles de operadores de Garlic se traducen entonces en operadores que el motor de ejecución puede ejecutar directamente.

El optimizador considera las funciones de planificación como reglas de enumeración. Las invoca para construir subplanes mediante dos funciones principales: `accessPlan`, para acceder a una relación, y `joinPlan`, para unir dos relaciones mediante los planes de acceso. Estas funciones reflejan con precisión las capacidades de los SGBD que los componen, con un formalismo común.

Ejemplo 7.19 Consideramos tres bases de datos de componentes, cada una en un sitio diferente. La base de datos de componentes db1 almacena la relación `EMP(ENO, ENAME, CITY)` y la base de datos de componentes db2 almacena la relación `WORKS(ENO, PNAME, DUR)`. La base de datos de componentes db3 almacena únicamente la información de los empleados con una única relación de esquema `EMPASG(ENAME, CITY, PNAME, DUR)`, cuya clave primaria es `(ENAME, PNAME)`. Las bases de datos de componentes db1 y db2 tienen el mismo contenedor w1, mientras que db3 tiene un contenedor diferente w2.

El contenedor w1 proporciona dos funciones de planificación típicas de un SGBD relacional. La regla `accessPlan`

```
accessPlan(R: relación, A: lista de atributos, P: predicado de selección)
      = scan(R, A, P , db(R))
```

produce un operador de escaneo que accede a tuplas de R desde su base de datos componente `db(R)` (aquí podemos tener `db(R) = db1 o db(R) = db2`), aplica el predicado de selección P y proyecta en la lista de atributos A. La regla `joinPlan`

```
joinPlan(R1, R2: relaciones, A: lista de atributos, P: predicado de unión) =
      join(R1, R2, A, P)
condición: db(R1) = db(R2)
```

Produce un operador de unión que accede a las tuplas de relaciones R1 y R2, aplica el predicado de unión P y proyecta la lista de atributos A. La condición indica que R1 y R2 se almacenan en bases de datos de componentes diferentes (es decir, db1 y db2). Por lo tanto, el operador de unión es implementado por el contenedor.

El contenedor w2 también proporciona dos funciones de planificación. La regla `accessPlan`

```
accessPlan(R: relación, A: lista de atributos, P: predicado de selección) =
      fetch(CIUDAD="c")
condición: (CIUDAD="c")    P
```

produce un operador de búsqueda que accede directamente a las tuplas de empleados (completas) en la base de datos de componentes db3 cuyo valor CITY es "c". La regla `accessPlan`

```
accessPlan(R: relación, A: lista de atributos, P: predicado de selección)
      = scan(R, A, P)
```

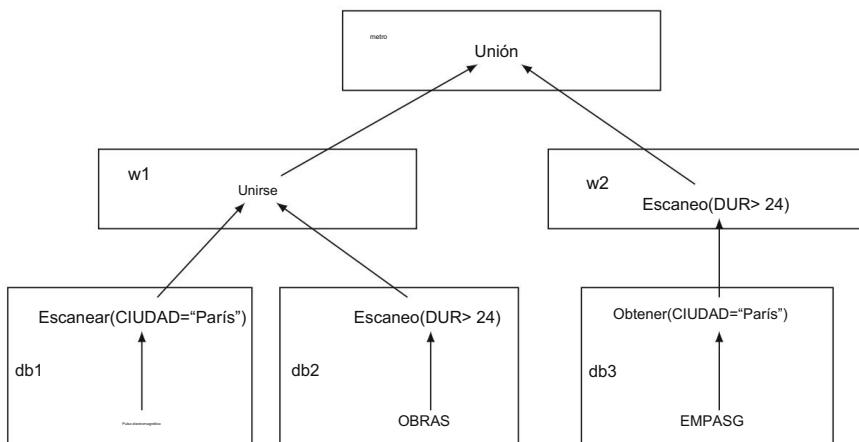


Fig. 7.15 Plan de ejecución de consultas heterogéneas

produce un operador de escaneo que accede a las tuplas de la relación R en el contenedor y aplica el predicado de selección P y la lista de proyectos de atributos A. Por lo tanto, el operador de escaneo lo implementa el contenedor, no el componente DBMS.

Consideré la siguiente consulta SQL enviada al mediador m:

```
SELECCIONAR ENAME, PNAME, DUR
DE EMPASG
DONDE CIUDAD = "París" Y DUR > 24
```

Asumiendo el enfoque GAV, la visión global EMPASG(ENAME, CITY, PNAME, DUR) se puede definir de la siguiente manera (para simplificar, anteponemos a cada relación el nombre de la base de datos de su componente):

```
EMPASG = (db1.EMP db2.WORKS) db3.EMPASG
```

Tras la reescritura de consultas en GAV y la optimización de las mismas, el enfoque basado en operadores podría generar el QEP que se muestra en la Fig. 7.15. Este plan muestra que los operadores no compatibles con el SGBD del componente deben ser implementados por los envoltorios o el mediador.

El uso de funciones de planificación para la optimización de consultas heterogéneas ofrece varias ventajas en los sistemas de bases de datos multivariables (MDBS). En primer lugar, las funciones de planificación proporcionan una forma flexible de expresar con precisión las capacidades de las fuentes de datos de los componentes. En particular, pueden utilizarse para modelar fuentes de datos no relacionales, como sitios web. En segundo lugar, dado que estas reglas son declarativas, facilitan el desarrollo de wrappers. El único desarrollo importante para los wrappers es la implementación de operadores específicos, por ejemplo, el operador de escaneo de db3 en el Ejemplo 7.19. Por último, este enfoque puede incorporarse fácilmente en un optimizador de consultas centralizado existente.

El enfoque basado en operadores también se ha utilizado con éxito en DIMDBS, un MDBS diseñado para acceder a múltiples bases de datos a través de la web. DISCO utiliza el

El enfoque GAV admite un modelo de datos de objetos para representar esquemas y tipos de datos de bases de datos de mediadores y componentes. Esto facilita la introducción de nuevas bases de datos de componentes, gestionando fácilmente posibles discrepancias de tipos. Las capacidades de los SGBD de componentes se definen como un subconjunto de una máquina algebraica (con los operadores habituales como scan, join y union) que puede ser soportada parcial o totalmente por los envoltorios o el mediador. Esto proporciona gran flexibilidad a los implementadores de envoltorios para decidir dónde soportar las capacidades de los SGBD de componentes (en el envoltorio o en el mediador). Además, se pueden especificar composiciones de operadores, incluyendo conjuntos de datos específicos, para reflejar las limitaciones de los SGBD de componentes. Sin embargo, el procesamiento de consultas es más complejo debido al uso de una máquina algebraica y a las composiciones de operadores. Tras la reescritura de consultas en los esquemas de componentes, hay tres pasos principales:

1. Generación del espacio de búsqueda. La consulta se descompone en varios QEP, lo que constituye el espacio de búsqueda para la optimización de consultas. Este espacio se genera mediante una estrategia de búsqueda tradicional, como la programación dinámica.
2. Descomposición de QEP. Cada QEP se descompone en un bosque de n QEP de envoltura y un QEP de composición. Cada QEP de envoltura constituye la parte más grande del QEP inicial que puede ser ejecutada completamente por el envoltorio. Los operadores que no pueden ser ejecutados por un envoltorio se trasladan al QEP de composición. El QEP de composición combina los resultados de los QEP de envoltura en la respuesta final, generalmente mediante uniones de los resultados intermedios producidos por los envoltorios.
3. Evaluación de costos. El costo de cada QEP se evalúa mediante un sistema de costos jerárquico. modelo discutido en la Secc. [7.2.4.1](#).

7.2.5 Traducción y ejecución de consultas

La traducción y ejecución de consultas se realiza mediante los wrappers que utilizan los SGBD de componentes. Un wrapper encapsula los detalles de una o más bases de datos de componentes, cada una compatible con el mismo SGBD (o sistema de archivos). También exporta al mediador las capacidades y funciones de coste del SGBD de componentes en una interfaz común. Uno de los principales usos prácticos de los wrappers ha sido permitir que un SGBD basado en SQL acceda a bases de datos no SQL.

La función principal de un contenedor es la conversión entre la interfaz común y la interfaz dependiente del SGBD. La Figura 7.16 muestra los diferentes niveles de interfaz entre el mediador, el contenedor y los SGBD que lo componen. Cabe destacar que, según el nivel de autonomía de los SGBD que lo componen, estos tres componentes pueden tener ubicaciones diferentes. Por ejemplo, en caso de una autonomía fuerte, el contenedor debería estar en la ubicación del mediador, posiblemente en el mismo servidor. Por lo tanto, la comunicación entre un contenedor y su SGBD que lo compone genera costos de red.

Sin embargo, en el caso de una base de datos de componentes cooperativa (por ejemplo, dentro de la misma organización), el contenedor podría instalarse en el sitio del DBMS del componente,

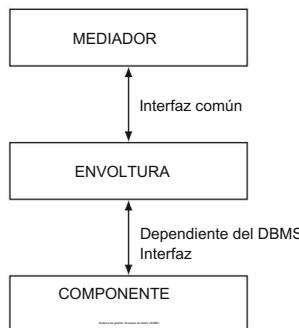


Fig. 7.16 Interfaces de envoltura

Como un controlador ODBC. Por lo tanto, la comunicación entre el contenedor y el SGBD del componente es mucho más eficiente.

La información necesaria para realizar la conversión se almacena en el esquema contenedor, que incluye el esquema local exportado al mediador en la interfaz común (p. ej., relacional) y las asignaciones de esquema para transformar datos entre el esquema local y el esquema de la base de datos del componente, y viceversa. Se analizaron las asignaciones de esquema en la sección 7.1.4. Se requieren dos tipos de conversión. En primer lugar, el contenedor debe traducir la entrada QEP generada por el mediador y expresada en una interfaz común en llamadas al SGBD del componente mediante su interfaz dependiente del SGBD.

Estas llamadas permiten la ejecución de consultas por parte del SGBD componente, que devuelve resultados expresados en la interfaz dependiente del SGBD. En segundo lugar, el contenedor debe traducir los resultados al formato de interfaz común para que puedan devolverse al mediador para su integración. Además, el contenedor puede ejecutar operaciones no compatibles con el SGBD componente (p. ej., la operación de escaneo del contenedor w2 en la figura 7.15).

Como se explicó en la sección 7.2.4.2, la interfaz común a los wrappers puede basarse en consultas o en operadores. El problema de la traducción es similar en ambos enfoques.

Para ilustrar la traducción de consultas en el siguiente ejemplo, utilizamos el enfoque basado en consultas con el estándar SQL/MED, que permite a un SGBD relacional acceder a datos externos representados como relaciones foráneas en el esquema local del contenedor. Este ejemplo ilustra cómo se puede encapsular una fuente de datos muy simple para acceder a ella mediante SQL.

Ejemplo 7.20 Consideramos la relación EMP(ENO, ENAME, CITY) almacenada en una base de datos de componentes muy simple, en el servidor ComponentDB, construida con archivos de texto Unix. Cada tupla EMP puede almacenarse como una línea en un archivo, por ejemplo, con los atributos separados por ":". En SQL/MED, la definición del esquema local para esta relación, junto con la asignación a un archivo Unix, puede declararse como una relación externa con la siguiente instrucción:

```

CREAR TABLA EXTRANJERA EMP
  ENO ENTERO,
  ENNAME VARCHAR(30),
  
```

```

    CIUDAD VARCHAR(20)
OPCIONES DE SERVIDOR
ComponentDB (Nombre de archivo '/usr/EngDB/emp.txt',
Delimitador ':')

```

Luego, el mediador puede enviar sentencias SQL al contenedor que admite el acceso a esta relación. Por ejemplo, la consulta

```

SELECCIONAR NOMBRE
DESDE EMP

```

El contenedor puede traducirlo utilizando el siguiente comando de shell de Unix para extraer el atributo relevante:

```
cortar -d: -f2 /usr/EngDB/emp
```

Luego se puede realizar un procesamiento adicional, por ejemplo para la conversión de tipos, usando código de programación.

Los wrappers se utilizan principalmente para consultas de solo lectura, lo que hace que la traducción de consultas y la construcción de wrappers sean relativamente fáciles. La construcción de wrappers generalmente se basa en herramientas con componentes reutilizables para generar la mayor parte del código del wrapper. Además, los proveedores de DBMS proporcionan wrappers para acceder de forma transparente a sus DBMS mediante interfaces estándar. Sin embargo, la construcción de wrappers es mucho más difícil si las actualizaciones de las bases de datos de componentes se deben soportar a través de wrappers (en lugar de actualizar directamente las bases de datos de componentes a través de sus DBMS). Un problema importante se debe a la heterogeneidad de las restricciones de integridad entre la interfaz común y la interfaz dependiente del DBMS. Como se discutió en el Cap. 3, las restricciones de integridad se utilizan para rechazar actualizaciones que violan la consistencia de la base de datos. En los DBMS modernos, las restricciones de integridad son explícitas y se especifican como reglas que forman parte del esquema de la base de datos. Sin embargo, en DBMS más antiguos o fuentes de datos más simples (por ejemplo, archivos), las restricciones de integridad son implícitas y se implementan mediante código específico en las aplicaciones. Por ejemplo, en el Ejemplo 7.20, podría haber aplicaciones con código incrustado que rechaza la inserción de nuevas líneas con una ENO existente en el archivo de texto EMP. Este código corresponde a una restricción de clave única en ENO en relación con EMP, pero no está disponible para el contenedor. Por lo tanto, el principal problema de la actualización mediante un contenedor es garantizar la consistencia de la base de datos de componentes, rechazando todas las actualizaciones que violan las restricciones de integridad, ya sean explícitas o implícitas. Una solución de ingeniería de software para este problema utiliza una herramienta con técnicas de ingeniería inversa para identificar dentro del código de la aplicación las restricciones de integridad implícitas, que luego se traducen en código de validación en los contenedores.

Otro problema importante es el mantenimiento del contenedor. La traducción de consultas depende en gran medida de las asignaciones entre el esquema de la base de datos de componentes y el esquema local. Si el esquema de la base de datos de componentes se modifica para reflejar la evolución de la base de datos de componentes, las asignaciones pueden volverse inválidas. Por ejemplo, en el Ejemplo 7.20, el administrador puede cambiar el orden de los campos en el archivo EMP. El uso de asignaciones inválidas puede impedir que el contenedor genere resultados correctos. Dado que

Las bases de datos de componentes son autónomas, por lo que detectar y corregir asignaciones no válidas es importante. Las técnicas para ello son las mismas que las de mantenimiento de asignaciones que se analizaron en este capítulo.

7.3 Conclusión

En este capítulo, analizamos el proceso de diseño ascendente de bases de datos, denominado integración de bases de datos, y cómo ejecutar consultas en bases de datos construidas de esta manera. La integración de bases de datos consiste en crear un GCS (o un esquema mediado) y determinar cómo se asigna cada LCS a él. Una distinción fundamental se da entre los almacenes de datos, donde el GCS se instancia y materializa, y los sistemas de integración de datos, donde el GCS es simplemente una vista virtual.

Aunque la integración de bases de datos se ha estudiado extensamente durante mucho tiempo, casi todo el trabajo ha estado fragmentado. Los proyectos individuales se centran en la coincidencia de esquemas, la limpieza de datos o el mapeo de esquemas. Se necesita una metodología integral para la integración de bases de datos que sea semiautomática y cuente con suficientes recursos para la participación de expertos. Un enfoque para dicha metodología es el trabajo de Bernstein y Melnik [2007], que proporciona los inicios de una metodología integral.

Un concepto relacionado que ha recibido considerable atención en la literatura es el intercambio de datos, definido como «el problema de tomar datos estructurados bajo un esquema fuente y crear una instancia de un esquema destino que refleje los datos fuente con la mayor precisión posible» [Fagin et al., 2005]. Esto es muy similar a la integración física (es decir, materializada) de datos, como los almacenes de datos, que analizamos en este capítulo. Una diferencia entre los almacenes de datos y los enfoques de materialización que se abordan en los entornos de intercambio de datos es que los datos del almacén de datos suelen pertenecer a una organización y pueden estructurarse según un esquema bien definido, mientras que en los entornos de intercambio de datos los datos pueden provenir de diferentes fuentes y presentar heterogeneidad.

En este capítulo nos hemos centrado en la integración de bases de datos. Sin embargo, cada vez más, los datos que se utilizan en aplicaciones distribuidas incluyen aquellos que no se encuentran en una base de datos. Un nuevo e interesante tema de debate entre los investigadores es la integración de datos estructurados almacenados en bases de datos y datos no estructurados mantenidos en otros sistemas (servidores web, sistemas multimedia, bibliotecas digitales, etc.). Los abordamos en el capítulo 12, donde nos centramos en la integración de datos de diferentes repositorios web y presentamos el concepto reciente de lagos de datos.

Otro problema que ignoramos en este capítulo es la integración de datos cuando no existe un GCS o no se puede especificar. Este problema surge especialmente en los sistemas peer-to-peer, donde la escala y la variedad de fuentes de datos dificultan (si no imposibilitan) el diseño de un GCS. Analizaremos la integración de datos en sistemas peer-to-peer en el capítulo 9.

La segunda parte de este capítulo se centró en el procesamiento de consultas en sistemas multibase de datos, que es significativamente más complejo que en sistemas de gestión de bases de datos distribuidos, homogéneos y estrechamente integrados. Además de ser distribuidas, las bases de datos de componentes pueden ser autónomas, tener diferentes lenguajes de base de datos y capacidades de procesamiento de consultas, y presentar un comportamiento variable. En particular, las bases de datos de componentes pueden abarcar desde bases de datos SQL completas hasta fuentes de datos muy simples (por ejemplo, archivos de texto).

En este capítulo, abordamos estos problemas ampliando y modificando la arquitectura de procesamiento distribuido de consultas presentada en el capítulo 4. Partiendo de la arquitectura popular de mediador/encapsulador, aislamos las tres capas principales mediante las cuales una consulta se reescribe sucesivamente (para aplicarla a las relaciones locales) y se optimiza mediante el mediador, para luego ser traducida y ejecutada por los encapsuladores y los SGBD que la componen. También analizamos cómo admitir consultas OLAP en una multibase de datos, un requisito importante para las aplicaciones de soporte de decisiones. Esto requiere una capa adicional de traducción de consultas OLAP multidimensionales a consultas relacionales. Esta arquitectura en capas para el procesamiento de consultas en multibase de datos es lo suficientemente general como para abarcar variaciones muy diversas. Esto ha sido útil para describir diversas técnicas de procesamiento de consultas, generalmente diseñadas con diferentes objetivos y supuestos.

Las principales técnicas para el procesamiento de consultas multibase de datos son la reescritura de consultas mediante vistas multibase de datos, la optimización y ejecución de consultas multibase de datos, y la traducción y ejecución de consultas. Las técnicas para la reescritura de consultas mediante vistas multibase de datos difieren de forma importante según se utilice el enfoque de integración GAV o LAV. La reescritura de consultas en GAV es similar a la localización de datos en sistemas de bases de datos distribuidas homogéneas. Sin embargo, las técnicas para LAV (y su extensión GLAV) son mucho más complejas y, a menudo, no es posible encontrar una reescritura equivalente para una consulta, en cuyo caso se necesita una consulta que produzca un subconjunto máximo de la respuesta. Las técnicas para la optimización de consultas multibase de datos incluyen el modelado de costes y la optimización de consultas para bases de datos componentes con diferentes capacidades informáticas. Estas técnicas amplían el procesamiento tradicional de consultas distribuidas centrándose en la heterogeneidad. Además de la heterogeneidad, un problema importante es abordar el comportamiento dinámico de los SGBD componentes. El procesamiento adaptativo de consultas aborda este problema con un enfoque dinámico mediante el cual el optimizador de consultas se comunica con el entorno de ejecución en tiempo de ejecución para reaccionar ante variaciones imprevistas en las condiciones de ejecución. Finalmente, analizamos las técnicas para traducir consultas para su ejecución por los componentes del SGBD y para generar y gestionar wrappers.

El modelo de datos utilizado por el mediador puede ser relacional, orientado a objetos u otros. En este capítulo, para simplificar, asumimos un mediador con un modelo relacional suficiente para explicar las técnicas de procesamiento de consultas multibase de datos. Sin embargo, al trabajar con fuentes de datos web, puede ser preferible un modelo de mediador más completo, como uno orientado a objetos o semiestructurado (p. ej., basado en XML o RDF). Esto requiere ampliaciones significativas de las técnicas de procesamiento de consultas.

7.4 Notas bibliográficas

Existe una amplia bibliografía sobre el tema de este capítulo. El trabajo se remonta a principios de la década de 1980 y ha sido bien analizado por Batini et al. [1986]. Elmagarmid et al. [1999] y Sheth y Larson [1990] abordan trabajos posteriores con gran detalle . Otra buena revisión más reciente del campo es la de Jhingran et al. [2002].

El libro de Doan et al. [2012] ofrece la cobertura más amplia del tema.

Existen varios artículos generales sobre el tema. Bernstein y Melnik [2007] ofrecen un análisis muy completo de la metodología de integración. Profundizan en la comparación del trabajo de gestión de modelos con algunas investigaciones sobre integración de datos.

Halevy et al. [2006] revisan el trabajo de integración de datos de la década de 1990, centrándose en el sistema Information Manifold [Levy et al. 1996a], que utiliza un enfoque LAV. El artículo proporciona una amplia bibliografía y analiza las áreas de investigación que se han abierto en los años transcurridos.

Haas [2007] adopta un enfoque integral para todo el proceso de integración y lo divide en cuatro fases: comprensión, que implica descubrir información relevante (claves, restricciones, tipos de datos, etc.), analizarla para evaluar su calidad y determinar las propiedades estadísticas; estandarización, mediante la cual se determina la mejor manera de representar la información integrada; especificación, que implica la configuración del proceso de integración; y ejecución, que es la integración propiamente dicha. La fase de especificación incluye las técnicas definidas en este artículo.

Los enfoques LAV y GAV son introducidos y analizados por Lenzerini [2002], Koch [2001] y Cali y Calvanese [2002]. El enfoque GLAV se analiza en [Friedman et al. 1999] y [Halevy 2001]. Se ha desarrollado un gran número de sistemas que han probado los enfoques LAV frente a GAV. Muchos de ellos se centran en la consulta sobre sistemas integrados. Se describen ejemplos de enfoques LAV en los artículos [Duschka y Genesereth 1997, Levy et al. 1996b, Manolescu et al. 2001], mientras que los ejemplos de GAV se presentan en artículos [Adali et al. 1996a, Garcia-Molina et al. 1997, Haas et al. 1997b].

Los temas de heterogeneidad estructural y semántica han ocupado a los investigadores durante bastante tiempo. Si bien la literatura sobre este tema es bastante extensa, algunas publicaciones interesantes que abordan la heterogeneidad estructural son [Dayal y Hwang 1984, Kim y Seo 1991, Breitbart et al. 1986, Krishnamurthy et al. 1991, Batini et al. 1986] (Batini et al. [1986] también analizan los conflictos estructurales introducidos en este capítulo) y aquellas que se centran en la heterogeneidad semántica son [Sheth y Kashyap 1992, Hull 1997, Ouksel y Sheth 1999, Kashyap y Sheth 1996, Bright et al. 1994, Ceri y Widom 1993, Vermeer 1997]. Cabe destacar que esta lista es bastante incompleta.

Existen diversas propuestas para el modelo canónico del GCS. Las que analizamos en este capítulo y sus fuentes son el modelo ER [Palopoli et al. 1998, Palopoli 2003, He y Ling 2006], el modelo orientado a objetos [Castano y Antonellis 1999, Bergamaschi 2001] y el modelo de grafos (que también se utiliza para determinar la similitud estructural) [Palopoli et al. 1999, Milo y Zohar 1998, Melnik et al. 2002].

Do y Rahm 2002, Madhavan et al. 2001], modelo trie [Madhavan et al. 2001] y XML [Yang et al. 2003].

Doan y Halevy [2005] ofrecen una excelente visión general de las diversas técnicas de emparejamiento de esquemas, proponiendo una clasificación diferente y más sencilla: basada en reglas, basada en aprendizaje y combinada. Rahm y Bernstein [2001] revisan más trabajos sobre emparejamiento de esquemas, lo que ofrece una comparación muy útil de diversas propuestas. Las reglas interesquemáticas que analizamos en este capítulo se deben a Palopoli et al. [1999]. La fuente clásica para las funciones de agregación de clasificación utilizadas en el emparejamiento es [Fagin 2002].

Se han desarrollado varios sistemas que demuestran la viabilidad de diversos enfoques de coincidencia de esquemas. Entre las técnicas basadas en reglas, se pueden citar DIKE [Palopoli et al. 1998, Palopoli 2003, Palopoli et al. 2003], DIPE, que es una versión anterior de este sistema [Palopoli et al. 1999], TranSCM [Milo y Zohar 1998], ARTEMIS [Bergamaschi 2001], inundación de similitud [Melnik et al. 2002], CUPID [Madhavan et al. 2001] y COMA [Do y Rahm 2002]. Para la coincidencia basada en el aprendizaje, Autoplex [Berlin y Motro 2001] implementa un clasificador bayesiano ingenuo, que también es el enfoque propuesto por Doan et al. [2001, 2003a] y Naumann et al. [2002]. En la misma clase, los árboles de decisión se analizan en [Embley et al. 2001, 2002] y iMAP en [Dhamankar et al. 2004].

Roth y Schwartz [1997], Tomasic et al. [1997] y Thiran et al. [2006] se centran en diversos aspectos de la tecnología de envoltorios. En [Thiran et al. 2006] se propone una solución de ingeniería de software para el problema de la creación y el mantenimiento de envoltorios, considerando el control de integridad.

Algunas fuentes para la integración binaria son [Batini et al. 1986, Pu 1988, Batini y Lenzirini 1984, Dayal y Hwang 1984, Melnik et al. 2002], mientras que los mecanismos n-arios se discuten en [Elmasri et al. 1987, Yao et al. 1982, He et al. 2004].

Para algunas herramientas de integración de bases de datos, los lectores pueden consultar [Sheth et al. 1988a], [Miller et al. 2001] que analizan Clio y [Roitman y Gal 2006] que describen OntoBuilder.

El algoritmo de creación de mapas de la sección 7.1.4.1 se debe a Miller et al. [2000], Yan et al. [2001] y [Popa et al. 2002]. El mantenimiento de mapas se analiza en Velegrakis et al. [2004].

La limpieza de datos ha cobrado gran interés en los últimos años a medida que los esfuerzos de integración se han abierto a fuentes de datos más ampliamente. La literatura sobre este tema es abundante y se aborda en detalle en el libro de Ilyas y Chu [2019]. En este contexto, la distinción entre limpieza a nivel de esquema y a nivel de instancia se debe a Rahm y Do [2000]. Los operadores de limpieza de datos que analizamos son la división de columnas [Raman y Hellerstein, 2001], el operador de mapa [Galhardas et al., 2001] y la coincidencia difusa [Chaudhuri et al., 2003].

El trabajo sobre el procesamiento de consultas multibase de datos comenzó a principios de la década de 1980 con los primeros sistemas multibase de datos (p. ej., [Brill et al. 1984, Dayal y Hwang 1984] y [Landers y Rosenberg 1982]). El objetivo entonces era acceder a diferentes bases de datos dentro de una organización. En la década de 1990, el creciente uso de la web para acceder a todo tipo de fuentes de datos despertó un renovado interés y una mayor investigación en el procesamiento de consultas multibase de datos, siguiendo la popular arquitectura mediador/encapsulador [Wiederhold].

[1992](#). Se puede encontrar una breve descripción general de los problemas de optimización de consultas multibase de datos en [Meng et al. [1993](#)]. Se pueden encontrar buenos análisis del procesamiento de consultas multibase de datos en [Lu et al. [1992, 1993](#)], en el capítulo 4 de [Yu y Meng [1998](#)] y en [Kossmann [2000](#)].

La reescritura de consultas mediante vistas se analiza en [Levy et al., [1995](#)] y se analiza en [Halevy, [2001](#)]. En [Levy et al., [1995](#)], se demuestra que el problema general de encontrar una reescritura mediante vistas es NP-completo en cuanto al número de vistas y subobjetivos de la consulta. La técnica de desdoblamiento para reescribir una consulta expresada en Datalog en GAV se propuso en [Ullman, [1997](#)]. Las principales técnicas para la reescritura de consultas mediante vistas en LAV son el algoritmo de cubo [Levy et al., [1996b](#)], el algoritmo de regla inversa [Duschka y Genesereth, [1997](#)] y el algoritmo MinCon [Pottinger y Levy, [2000](#)].

Los tres enfoques principales para el modelado de costos heterogéneos se discuten en [Zhu y Larson [1998](#)]. El enfoque de caja negra se utiliza en [Du et al. [1992](#), Zhu y Larson [1994](#)]; las técnicas en este grupo son consultas de sondeo: [Zhu y Larson [1996a](#)], consultas de muestra (que son un caso especial de sondeo) [Zhu y Larson [1998](#)], y aprender el costo con el tiempo a medida que las consultas se plantean y responden [Adali et al. [1996b](#)]. El enfoque personalizado se desarrolla en [Zhu y Larson [1996b](#), Roth et al. [1999](#), Naacke et al. [1999](#)]; en particular, el cálculo de costos se puede hacer dentro del envoltorio (como en Garlic) [Roth et al. [1999](#)] o se puede desarrollar un modelo de costos jerárquico (como en Disco) [Naacke et al. [1999](#)]. El enfoque dinámico se utiliza en [Zhu et al. [2000](#)], [Zhu et al. [2003](#)] y [Rahal et al. [2004](#)], y también lo analizan Lu et al. [\[1992\]](#). Zhu [\[1995\]](#) analiza un muestreo con enfoque dinámico, y Zhu et al. [\[2000\]](#) presentan un enfoque cualitativo.

El algoritmo que describimos para ilustrar el enfoque basado en consultas para la optimización de consultas heterogéneas (Sección 7.2.4.2) se propuso en [Du et al., [1995](#)] y también se analizó en [Evrendilek et al., [1997](#)]. Para ilustrar el enfoque basado en operadores, describimos la solución popular con funciones de planificación propuesta en el proyecto Garlic [Haas et al., [1997a](#)]. Este enfoque también se utilizó en DISCO, un sistema multibase de datos para acceder a bases de datos de componentes a través de la web [Tomasic et al., [1996, 1998](#)].

El caso del procesamiento adaptativo de consultas ha sido presentado por varios investigadores en varios entornos. Amsaleg et al. [\[1996\]](#) muestra por qué los planes estáticos no pueden lidiar con la imprevisibilidad de las fuentes de datos; el problema existe en consultas continuas [Madden et al. [2002b](#)], predicados costosos [Porto et al. [2003](#)] y sesgo de datos [Shah et al. [2003](#)]. El enfoque adaptativo se examina en [Hellerstein et al. [2000](#), Gounaris et al. [2002](#)]. El enfoque dinámico más conocido es eddy (ver Cap. 4), que se analiza en [Avnur y Hellerstein [2000](#)]. Otras técnicas importantes para el procesamiento adaptativo de consultas son la codificación de consultas [Amsaleg et al. [1996](#), Urhan et al. [1998](#)], las uniones de ondulación [Haas y Hellerstein [1999b](#)] y el particionamiento adaptativo [Shah et al. [2003](#)] y Cherry picking [Porto et al. [2003](#)]. Las principales extensiones de eddy son los módulos de estado [Raman et al. [2003](#)] y los Eddies distribuidos [Tian y DeWitt [2003](#)].

En este capítulo, nos centramos en la integración de datos estructurados capturados en bases de datos. El problema más general de la integración de datos estructurados y no estructurados es abordado por Halevy et al. [\[2003\]](#) y Somani et al. [\[2002\]](#). Un enfoque diferente

Bernstein y Melnik [2007] investigan la dirección de generalidad y proponen un motor de gestión de modelos que “admite operaciones para hacer coincidir esquemas, componer mapeos, diferenciar esquemas, fusionar esquemas, traducir esquemas en diferentes modelos de datos y generar transformaciones de datos a partir de mapeos”.

Además de los sistemas mencionados anteriormente, en este capítulo nos referimos a otros sistemas. Estos y sus principales fuentes son: SEMINT [Li y Clifton 2000, Li et al. 2000], ToMAS [Velegrakis et al. 2004], Maveric [McCann et al. 2005] y Aurora [Yan 1997, Yan et al. 1997].

Ceremonias

Problema 7.1 Los sistemas de bases de datos distribuidas y los sistemas multibase de datos distribuidos representan dos enfoques diferentes para el diseño de sistemas. Encuentre tres aplicaciones reales para las que cada uno de estos enfoques sea más apropiado. Analice las características de estas aplicaciones que las hacen más favorables para uno u otro enfoque.

Problema 7.2 Algunos modelos arquitectónicos favorecen la definición de un esquema conceptual global, mientras que otros no. ¿Qué opina? Justifique su elección con argumentos técnicos detallados.

Problema 7.3 (*) Dé un algoritmo para convertir un esquema relacional a uno entidad-relación.

Problema 7.4 ()** Considere las dos bases de datos que se muestran en las figuras 7.17 y 7.18 y que se describen a continuación. Diseñe un esquema conceptual global como unión de las dos bases de datos, traduciéndolas primero al modelo ER.

La Figura 7.17 describe una base de datos de carreras relacionales utilizada por organizadores de carreras de ruta y la Figura 7.18 describe una base de datos entidad-relación utilizada por un fabricante de calzado. A continuación se describe la semántica de cada uno de estos esquemas de base de datos. La Figura 7.17 describe una base de datos relacional de carreras de ruta con la siguiente semántica:

DIRECTOR es una relación que define a los directores de carrera que organizan las carreras; asumimos que cada director de carrera tiene un nombre único (que se utilizará como clave), un número de teléfono y una dirección.

```

DIRECTOR(NAME, PHONE_NO, ADDRESS)
LICENSES(LIC_NO, CITY, DATE, ISSUES, COST, DEPT, CONTACT)
RACER(NAME, ADDRESS, MEM_NUM)
SPONSOR(SP_NAME, CONTACT)
RACE(R_NO, LIC_NO, DIR, MAL_WIN, FRM_WIN, SP_NAME)

```

Fig. 7.17 Base de datos de carreras en ruta

Se requieren LICENCIAS porque todas las razas requieren una licencia gubernamental, que es emitida por un CONTACTO en un departamento que es el EMISOR, posiblemente contenido dentro de otro departamento gubernamental DEPT; cada licencia tiene un LIC_NO único (la clave), que se emite para su uso en una CIUDAD específica en una FECHA específica con un COSTO determinado.

RACER es una relación que describe a las personas que participan en una carrera. Cada persona se identifica por su NOMBRE, lo cual no es suficiente para identificarla de forma única, por lo que se requiere una clave compuesta formada con la DIRECCIÓN. Finalmente, cada corredor puede tener un MEM_NUM para identificarse como miembro de la fraternidad de carreras, pero no todos los competidores tienen números de membresía.

PATROCINADOR indica qué patrocinador financia una carrera determinada. Normalmente, un patrocinador financia varias carreras a través de una persona específica (CONTACTO), y varias carreras pueden tener diferentes patrocinadores.

RACE identifica de forma única una única carrera que tiene un número de licencia (LIC_NO) y un número de carrera (R_NO) (para usarse como clave, ya que se puede planificar una carrera sin adquirir aún una licencia); cada carrera tiene un ganador en los grupos masculino y femenino (MAL_WIN y FEM_WIN) y un director de carrera (DIR).

La figura 7.18 ilustra un esquema entidad-relación utilizado por el sistema de base de datos del patrocinador con la siguiente semántica:

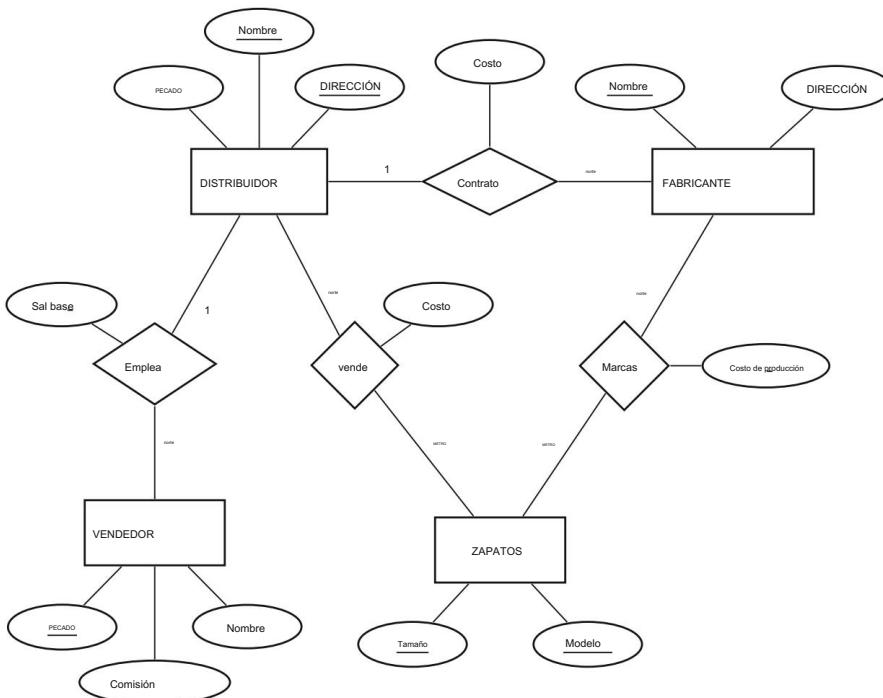


Fig. 7.18 Base de datos de patrocinadores

Los ZAPATOS son producidos por patrocinadores de un MODELO y TALLA determinados, lo que forma la clave de la entidad.

EL FABRICANTE se identifica de forma única por NOMBRE y reside en un lugar determinado DIRECCIÓN.

DISTRIBUIDOR es una persona que tiene NOMBRE y DIRECCIÓN (que son necesario para formar la clave) y un número SIN para efectos fiscales.

VENDEDOR es una persona (entidad) que tiene un NOMBRE, gana una COMISIÓN, y se identifica de forma única por su número SIN (la clave).

Makes es una relación que tiene un cierto costo de producción fijo (PROD_COST).

Indica que un fabricante fabrica varios zapatos diferentes y que distintos fabricantes producen el mismo zapato.

Sells es una relación que indica el COSTO al por mayor a un distribuidor de zapatos.

Indica que cada distribuidor vende más de un tipo de zapato, y que cada tipo de zapato es vendido por más de un distribuidor.

El contrato es una relación mediante la cual un distribuidor adquiere, a cambio de un COSTO, derechos exclusivos para representar a un fabricante. Cabe destacar que esto no le impide vender calzado de otros fabricantes.

Employ es una relación que indica que cada distribuidor contrata una cantidad de vendedores para vender los zapatos; cada uno gana un SALARIO_BASE.

Problema 7.5 (*) Considere tres fuentes:

- La base de datos 1 tiene una relación Área(Id, Campo) que proporciona áreas de especialización.ción de empleados; el campo Id identifica a un empleado.

La base de datos 2 tiene dos relaciones: Enseñar(Profesor, Curso) e In(Curso, Campo); Enseñar indica los cursos que imparte cada profesor e In especifica los campos a los que puede pertenecer un curso. La base de datos 3 tiene dos relaciones: Beca(Investigador, N.^o de Beca) para las becas otorgadas a investigadores, y Para(N.^o de Beca, Campo) para los campos a los que corresponden las becas.

El objetivo es construir un GCS con dos relaciones: Trabajos(Id, Proyecto) que indica que un empleado trabaja para un proyecto en particular, y Área(Proyecto, Campo) que asocia proyectos con uno o más campos.

(a) Proporcione un mapeo LAV entre la Base de Datos 1 y el GCS. (b)

Proporcione un mapeo GLAV entre el GCS y los esquemas locales. (c) Suponga que se agrega una relación adicional, Fondos(GrantNo, Proyecto), a la Base de Datos 3. Proporcione un mapeo GAV en este caso.

Problema 7.6. Considere un GCS con la siguiente relación: Persona(Nombre, Edad, Género). Esta relación se define como una vista de tres LCS de la siguiente manera:

```

CREAR VISTA Persona COMO
SELECCIONE Nombre, Edad, "masculino" AS Género
DE SoccerPlayer
UNIÓN
SELECCIONAR Nombre, NULL AS Edad, Género
  
```

```

DE Actor
UNIÓN
SELECCIONAR Nombre, Edad, Género
DE Político
DONDE Edad > 30

```

Para cada una de las siguientes consultas, analice cuál de los tres esquemas locales (SoccerPlayer, Actor y Político) contribuye al resultado de la consulta global.

- (a) SELECCIONAR Nombre DE Persona
- (b) SELECCIONAR Nombre DE Persona DONDE Género = "femenino"
- (c) SELECCIONAR Nombre DE Persona DONDE Edad > 25
- (d) SELECCIONAR Nombre DE Persona DONDE Edad < 25
- (e) SELECCIONAR Nombre DE Persona DONDE Género = "masculino" Y Edad = 40

Problema 7.7 Un GCS con la relación País(Nombre, Continente, Población, TieneCosta) describe países del mundo. El atributo TieneCosta indica si el país tiene acceso directo al mar. Tres LCS se conectan al esquema global mediante el enfoque LAV, como se indica a continuación:

```

CREAR VISTA EuropeanCountry AS
SELECCIONAR Nombre, Continente, Población, TieneCosta
DESDE País
DONDE Continente = "Europa"

```

```

CREAR VISTA BigCountry AS
SELECCIONAR Nombre, Continente, Población, TieneCosta
DESDE País
DONDE Población >= 30000000

```

```

CREAR VISTA MidsizeOceanCountry AS
SELECCIONAR Nombre, Continente, Población, TieneCosta
DESDE País
DONDE HasCoast = verdadero Y Población > 10000000

```

- (a) Para cada una de las siguientes consultas, analice los resultados con respecto a su integridad, es decir, verifique si la (combinación de) fuentes locales cubre todos los resultados relevantes.

1. SELECCIONE Nombre DE País 2.

```

SELECCIONE Nombre DE País DONDE Población > 40 3. SELECCIONE
Nombre DE País DONDE Población > 20

```

- (b) Para cada una de las siguientes consultas, analice cuáles de las tres LCS son necesarias para el resultado de la consulta global.

1. SELECCIONE Nombre DE País

2. SELECCIONE Nombre DE País DONDE Población > 30
Y Continente = "Europa"
3. SELECCIONE Nombre DE País DONDE Población < 30
4. SELECCIONE Nombre DE País DONDE Población > 30
Y HasCoast = verdadero

Problema 7.8 Considere las siguientes dos relaciones PRODUCTO y ARTÍCULO que Se especifican en una notación SQL simplificada. Las correspondencias de esquema perfectas se indican con flechas.

PRODUCTO	→ ARTÍCULO
Id: int CLAVE PRIMARIA	→ Clave: varchar(255) CLAVE PRIMARIA
Nombre: varchar(255)	→ Título: varchar(255)
Precio de entrega: flotante	→ Precio: real
Descripción: varchar(8000)	→ Información: varchar(5000)

(a) Para cada una de las cinco correspondencias, indique cuál de las siguientes corresponde

Los enfoques probablemente identificarán la correspondencia:

1. Comparación sintáctica de nombres de elementos, por ejemplo, utilizando la cadena de distancia de edición semejanza
2. Comparación de nombres de elementos utilizando una tabla de búsqueda de sinónimos
3. Comparación de tipos de datos
4. Análisis de valores de datos de instancia

(b) ¿Es posible que los métodos de correspondencia enumerados determinen correspondencias falsas?

¿Cuáles son las ventajas de estas tareas de emparejamiento? Si es así, dé un ejemplo.

Problema 7.9 Considere dos relaciones S(a, b, c) y T(d, e, f). Una coincidencia

El enfoque determina las siguientes similitudes entre los elementos de S y T:

	Td	Te	Tf
Sa 0.8		0.3	0.1
Sb 0,5 0,2 0,9			
Sc 0,4 0,7 0,8			

Con base en el resultado del comparador proporcionado, obtenga un resultado de coincidencia de esquema general con las siguientes características:

- Cada elemento participa en exactamente una correspondencia.
- No existe correspondencia donde ambos elementos coincidan con un elemento del esquema opuesto con una similitud mayor que su contraparte correspondiente.

Problema 7.10 (*) La figura 7.19 ilustra el esquema de tres datos diferentes fuentes:

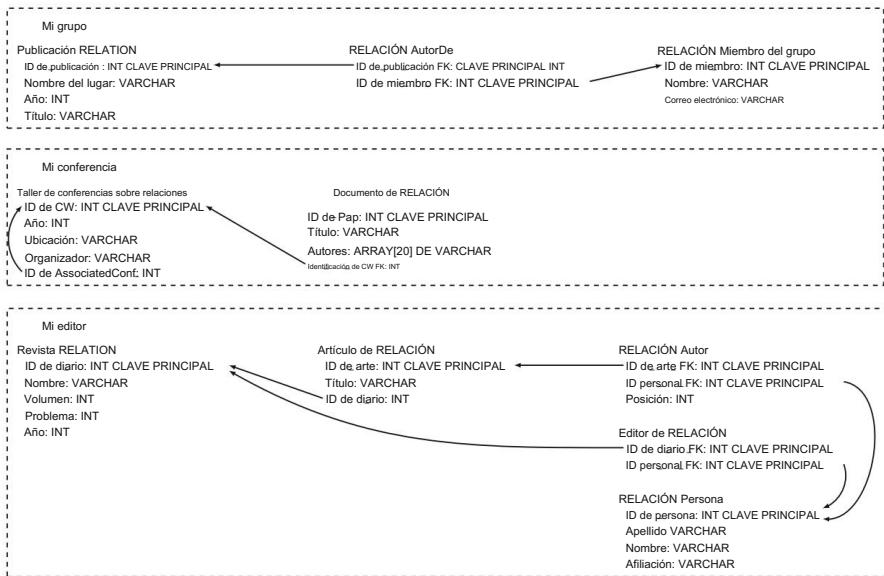


Fig. 7.19 Figura para el ejercicio 7.10

- MyGroup contiene publicaciones creadas por miembros de un grupo de trabajo;
- MyConference contiene publicaciones de una serie de conferencias y publicaciones asociadas. talleres;
- MyPublisher contiene artículos que se publican en revistas.

Las flechas muestran las relaciones entre la clave externa y la clave principal; tenga en cuenta que no seguimos la sintaxis SQL adecuada para especificar relaciones de clave externa para ahorrar espacio; recurrimos a las flechas.

Las fuentes se definen de la siguiente manera:

Mi grupo

- Publicación

- Pub_ID: ID de publicación único
- VenueName: nombre de la revista, conferencia o taller
- VenueType: "revista", "conferencia" o "taller"
- Year: año de publicación
- Title: título de la publicación

- Autor de

- relación de muchos a muchos que representa "el miembro del grupo es el autor de la publicación"

- Miembro del grupo

- Member_ID: ID de miembro único

- Nombre: nombre del miembro del grupo • Correo electrónico: dirección de correo electrónico del miembro del grupo

Mi conferencia

- Taller de conferencias
 - CW_ID: ID único para la conferencia/taller • Nombre: nombre de la conferencia o taller • Año: año en que se lleva a cabo el evento • Ubicación: ubicación del evento • Organizador: nombre de la persona organizadora • AssociatedConf_ID_FK: el valor es NULL si es una conferencia, ID de la conferencia asociada si el evento es un taller (esto supone que los talleres se organizan junto con una conferencia)

• Papel

- Pap_ID: ID único del artículo • Título: título del artículo • Autor: matriz de nombres de autores • CW_ID_FK: conferencia/taller donde se publicó el artículo

Mi editor

- Diario
 - Journ_ID: ID único de la revista • Nombre: nombre de la revista • Año: año en que se lleva a cabo el evento • Volumen: volumen de la revista • Número: número de la revista
- Artículo
 - Art_ID: ID único del artículo • Título: título del artículo • Journ_ID_FK: revista donde se publica el artículo
- Persona
 - Pers_ID: ID único de la persona • LastName: apellido de la persona • FirstName: nombre de la persona • Affiliation: afiliación de la persona (por ejemplo, el nombre de una universidad)
- Autor
 - representa la relación de muchos a muchos para “la persona es el autor del artículo” • Posición: posición del autor en la lista de autores (por ejemplo, el primer autor tiene la posición 1)
- Editor

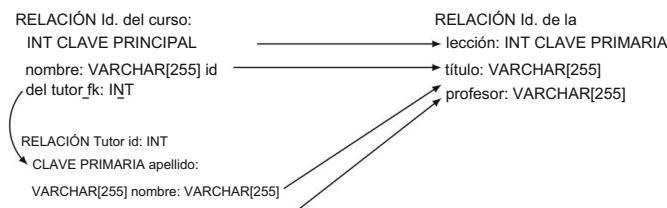


Fig. 7.20 Figura para el ejercicio 7.11

- representa la relación de muchos a muchos para "la persona es editor de la revista asunto"

(a) Identifique todas las correspondencias de esquema entre los elementos de esquema de las fuentes.

Utilice los nombres y tipos de datos de los elementos de esquema, así como la descripción dada.

(b) Clasifique sus

correspondencias según las siguientes dimensiones:

1. Tipo de elementos del esquema (por ejemplo, atributo-atributo o atributo-relación)
2. Cardinalidad (por ejemplo, 1:1 o 1:N)

(c) Proporcione un esquema global consolidado que cubra toda la información de la fuente.

esquemas.

Problema 7.11 (*) La Figura 7.20 ilustra (utilizando una sintaxis SQL simplificada) dos fuentes: Fuente1 y Fuente2. Fuente1 tiene dos relaciones: Curso y Tutor, y Fuente2 solo tiene una relación: Clase. Las flechas sólidas indican correspondencias de esquema. La flecha discontinua representa una relación de clave externa entre las dos relaciones en Fuente1.

Las siguientes son cuatro asignaciones de esquema (representadas como consultas SQL) para transformar los datos de Source1 en Source2.

1. SELECCIONE C.id, C.nombre como Título, CONCAT(T.apellido,
T.firstname) Profesor AS

DESDE Curso COMO C

ÚNETE al Tutor AS T ON (C.tutor_id_fk = T.id)

2. SELECCIONE C.id, C.name COMO Título, NULL COMO Profesor

DESDE Curso COMO C

UNIÓN

SELECCIONAR T.id COMO ID, NULO COMO Título, T,
Apellido AS Profesor

DESDE Curso COMO C

UNIÓN EXTERNA COMPLETA Tutor AS T ON(C.tutor_id_fk=T.id)

3. SELECCIONE C.id, C.nombre como Título, CONCAT(T.apellido,

T.firstname) Profesor AS

DESDE Curso COMO C

UNIÓN EXTERNA COMPLETA Tutor AS T ON(C.tutor_id_fk=T.id)

Analice cada una de estas asignaciones de esquema con respecto a las siguientes preguntas:

(a) ¿Es significativa la asignación? (b)

¿Es completa la asignación (es decir, se transforman todas las instancias de datos de O1)? (c)

¿La asignación viola potencialmente las restricciones clave?

Problema 7.12 (*) Considere tres fuentes de datos:

La base de datos 1 tiene una relación AREA(ID, CAMPO) que proporciona las áreas de especialización de los empleados, donde ID identifica a cada empleado. La base de datos 2 tiene dos relaciones: TEACH(PROFESOR, CURSO) y IN(CURSO, CAMPO), que especifican los campos a los que puede pertenecer un curso. La base de datos 3 tiene dos relaciones: GRANT(INVESTIGADOR, NÚMERO DE BECA) para las becas otorgadas a investigadores, y FOR(NÚMERO DE BECA, CAMPO), que indica los campos a los que pertenecen las becas.

Diseñe un esquema global con dos relaciones: WORKS(ID, PROJECT) que registre en qué proyectos trabajan los empleados y AREA(PROJECT, FIELD) que asocie proyectos con uno o más campos para los siguientes casos:

(a) Debe haber una asignación LAV entre la Base de datos 1 y el esquema global. (b) Debe haber una asignación GLAV entre el esquema global y el esquema local.

esquemas.

(c) Debería haber una asignación GAV cuando se agrega una relación adicional FONDOS(SUCURSAL#, PROYECTO) se agrega a la base de datos 3.

Problema 7.13 (**) La lógica (lógica de primer orden, para ser más precisos) se ha propuesto como un formalismo uniforme para la traducción e integración de esquemas. Analice cómo la lógica puede ser útil para este propósito.

Problema 7.14 (**) ¿Es posible realizar cualquier tipo de optimización global en consultas globales en un sistema multibase de datos? Analice y especifique formalmente las condiciones bajo las cuales dicha optimización sería posible.

Problema 7.15 (**) Considere las relaciones globales EMP(ENAME, TITLE, CITY) y ASG(ENAME, PNAME, CITY, DUR). CITY en ASG es la ubicación del proyecto de nombre PNAME (es decir, PNAME determina funcionalmente CITY). Considere las relaciones locales EMP1(ENAME, TITLE, CITY), EMP2(ENAME, TITLE, CITY), PROJ1(PNAME, CITY), PROJ2(PNAME, CITY) y ASG1(ENAME, PNAME, DUR). Considere la consulta Q que selecciona los nombres de los empleados asignados a un proyecto en Río de Janeiro por más de 6 meses y la duración de su asignación.

(a) Suponiendo el enfoque GAV, realice la reescritura de la consulta. (b)

Suponiendo el enfoque LAV, realice la reescritura de la consulta utilizando el algoritmo bucket. ritmo.

(c) Igual que (b) utilizando el algoritmo MinCon.

Problema 7.16 (*) Considérense las relaciones EMP y ASG del Ejemplo 7.18. Denotamos por $|R|$ el número de páginas que se almacenarán en el disco. Considera las siguientes estadísticas sobre Los datos:

$$|\text{EMP}| = 100$$

$$|\text{ASG}| = 2\,000$$

$$\text{selectividad}(\text{ASG.DUR} > 36) = 1\%$$

El modelo de coste genérico del mediador es

$$\text{costo } (\sigma A=v(R)) = |R|$$

coste ($\sigma (X)$) = coste (X), donde X contiene al menos un operador.

$$\text{costo } (RS) = \underset{\text{indexada}}{\text{costo}} (R) + |R| \quad \text{costo } (\sigma A=v(S)) \text{ utilizando un algoritmo de unión indexada.}$$

$$\text{costo } (R \underset{\text{Anulado}}{\text{Un n.}} S) = \text{costo } (R) + |R| \quad \text{costo } (S) \text{ utilizando un algoritmo de unión de bucle anidado.}$$

Considera la consulta de entrada MDBS Q:

```
SELECCIONAR *
DESDE EMP NATURAL ÚNETE A ASG
DONDE ASG.DUR>36
```

Considera cuatro planes para procesar Q:

P1 = EMP	<small>Indiana</small>	ENO	$\sigma_{DUR>36}(\text{ASG})$
P2 = EMP	<small>nl</small>	ENO	$\sigma_{DUR>36}(\text{ASG})$
P3 = $\sigma_{DUR>36}(\text{ASG})$	<small>Indiana</small>	ENO	<small>Para el escaneamiento</small>
P4 = $\sigma_{DUR>36}(\text{ASG})$	<small>nl</small>	ENO	<small>Para el escaneamiento</small>

(a) ¿Cuál es el costo de los planes P1 a P4?

(b) ¿Qué plan tiene el costo mínimo?

Problema 7.17 (*) Consideremos las relaciones EMP y ASG del ejercicio anterior.

Supongamos ahora que el modelo de costos del mediador se completa con el siguiente costo información emitida desde los DBMS componentes.

El costo de acceder a las tuplas EMP en db1 es

$$\text{costo } (\sigma A=v(R)) = |\sigma A=v(R)|$$

El costo específico de seleccionar tuplas ASG que tienen un ENO determinado en db2 es

$$\text{costo } (\sigma \text{ENO}=v(\text{ASG})) = |\sigma \text{ENO}=v(\text{ASG})|$$

(a) ¿Cuál es el costo de los planes P1 a P4?

(b) ¿Qué plan tiene el costo mínimo?

Problema 7.18 (**). ¿Cuáles son las respectivas ventajas y limitaciones de los enfoques basados en consultas y operadores para la optimización de consultas heterogéneas desde los puntos de vista de la expresividad de la consulta, el rendimiento de la consulta, el costo de desarrollo de envoltorios, el mantenimiento del sistema (mediador y envoltorios) y la evolución?

Problema 7.19 (**). Considere el Ejemplo 7.19 añadiendo, en un nuevo sitio, la base de datos de componentes db4 , que almacena las relaciones EMP(ENO, ENAME, CITY) y ASG(ENO, PNAME, DUR). db4 exporta mediante sus capacidades de unión y escaneo de contenedor w3 . Supongamos que puede haber empleados en db1 con asignaciones correspondientes en db4 y empleados en db4 con asignaciones correspondientes en db2.

(a) Defina las funciones de planificación del contenedor w3. (b)

Dé la nueva definición de la vista global EMPASG(ENAME, CITY, PNAME,
DURACIÓN).

(c) Proporcione un QEP para la misma consulta que en el Ejemplo 7.19.

Capítulo 8

Sistemas de bases de datos paralelas



Muchas aplicaciones con uso intensivo de datos requieren soporte para bases de datos muy grandes (p. ej., cientos de terabytes o exabytes). El soporte eficiente de bases de datos muy grandes, tanto para OLTP como para OLAP, se puede lograr combinando computación paralela y gestión distribuida de bases de datos.

Una computadora paralela, o multiprocesador, es un tipo de sistema distribuido compuesto por varios nodos (procesadores, memorias y discos) conectados mediante una red de alta velocidad dentro de uno o más gabinetes en la misma habitación. Existen dos tipos de multiprocesadores según cómo estén acoplados estos nodos: estrechamente acoplados y débilmente acoplados. Los multiprocesadores estrechamente acoplados contienen múltiples procesadores conectados a nivel de bus con memoria compartida. Las computadoras mainframe, las supercomputadoras y los procesadores multinúcleo modernos utilizan el acoplamiento estrecho para optimizar el rendimiento. Los multiprocesadores débilmente acoplados, ahora conocidos como clústeres de computadoras, o clústeres para abreviar, se basan en múltiples computadoras comunes interconectadas mediante una red de alta velocidad. La idea principal es construir una computadora potente a partir de muchos nodos pequeños, cada uno con una excelente relación costo-rendimiento, a un costo mucho menor que las computadoras mainframe o supercomputadoras equivalentes. En su forma más económica, la interconexión puede ser una red local. Sin embargo, ahora existen interconexiones estándar rápidas para clústeres (por ejemplo, Infiniband y Myrinet) que proporcionan un gran ancho de banda (por ejemplo, 100 Gigabits/seg) con baja latencia para el tráfico de mensajes.

Como ya se discutió en capítulos anteriores, la distribución de datos se puede aprovechar para aumentar el rendimiento (a través del paralelismo) y la disponibilidad (a través de la replicación). Este principio se puede utilizar para implementar sistemas de bases de datos paralelas, es decir, sistemas de bases de datos en computadoras paralelas. Los sistemas de bases de datos paralelas pueden aprovechar el paralelismo en la gestión de datos para ofrecer servidores de bases de datos de alto rendimiento y alta disponibilidad. Por lo tanto, pueden soportar bases de datos muy grandes con cargas muy elevadas.

La mayor parte de la investigación sobre sistemas de bases de datos paralelas se ha realizado en el contexto del modelo relacional, ya que proporciona una buena base para el procesamiento paralelo de datos. En este capítulo, presentamos el enfoque de sistemas de bases de datos paralelas como una solución para la gestión de datos de alto rendimiento y alta disponibilidad. Analizamos sus ventajas y

Desventajas de las diversas arquitecturas de sistemas paralelos y presentamos las técnicas de implementación genéricas.

La implementación de sistemas de bases de datos paralelas se basa naturalmente en técnicas de bases de datos distribuidas. Sin embargo, los aspectos críticos son la ubicación de los datos, el procesamiento de consultas paralelas y el equilibrio de carga, ya que el número de nodos puede ser mucho mayor que el número de sitios en un SGBD distribuido. Además, una computadora paralela suele proporcionar una comunicación fiable y rápida que puede aprovecharse para implementar eficientemente la gestión y replicación de transacciones distribuidas. Por lo tanto, aunque los principios básicos son los mismos que en los SGBD distribuidos, las técnicas para los sistemas de bases de datos paralelas son bastante diferentes.

Este capítulo se organiza de la siguiente manera: en la sección 8.1, se aclaran los objetivos de los sistemas de bases de datos paralelas. En la sección 8.2, se analizan las arquitecturas, en particular las de memoria compartida, disco compartido y sin recursos compartidos. A continuación, se presentan las técnicas de ubicación de datos (sección 8.3), el procesamiento de consultas (sección 8.4), el balanceo de carga (sección 8.5) y la tolerancia a fallos (sección 8.6). En la sección 8.7, se presenta el uso de técnicas de gestión de datos paralelos en clústeres de bases de datos, un tipo importante de sistema de bases de datos paralelas.

8.1 Objetivos

El procesamiento paralelo aprovecha las computadoras multiprocesador para ejecutar programas de aplicación mediante el uso conjunto de varios procesadores y así mejorar el rendimiento. Su uso principal se ha centrado desde hace tiempo en la computación científica para mejorar el tiempo de respuesta de las aplicaciones numéricas. Los avances en computadoras paralelas de propósito general que utilizan microprocesadores estándar y técnicas de programación paralela han permitido la irrupción del procesamiento paralelo en el campo del procesamiento de datos.

Los sistemas de bases de datos paralelas combinan la gestión y el procesamiento paralelo de bases de datos para aumentar el rendimiento y la disponibilidad. Cabe destacar que el rendimiento también fue el objetivo de las máquinas de bases de datos en la década de 1980. El problema al que se enfrentaba la gestión convencional de bases de datos se conoce desde hace tiempo como "cuello de botella de E/S", provocado por un alto tiempo de acceso al disco con respecto al tiempo de acceso a la memoria principal (normalmente cientos de miles de veces más rápido). Inicialmente, los diseñadores de máquinas de bases de datos abordaron este problema mediante hardware específico, por ejemplo, introduciendo dispositivos de filtrado de datos en los cabezales de disco. Sin embargo, este enfoque fracasó debido a su bajo coste/rendimiento en comparación con la solución de software, que puede beneficiarse fácilmente de los avances del hardware en la tecnología de silicio. La idea de acercar las funciones de las bases de datos al disco ha cobrado un renovado interés con la introducción de microprocesadores de propósito general en los controladores de disco, lo que ha dado lugar a los discos inteligentes. Por ejemplo, las funciones básicas que requieren un costoso escaneo secuencial, como las operaciones de selección en tablas con predicados difusos, pueden ejecutarse de forma más eficiente a nivel de disco, ya que evitan la sobrecarga de la memoria del SGBD con bloques de disco irrelevantes. Sin embargo, la explotación de discos inteligentes requiere adaptar el DBMS, en particular, el procesador de

Para usar las funciones del disco. Dado que no existe una tecnología de disco inteligente estándar, la adaptación a diferentes tecnologías de disco inteligente perjudica la portabilidad del SGBD.

Sin embargo, un resultado importante reside en la solución general al cuello de botella de E/S. Esta solución se puede resumir en el aumento del ancho de banda de E/S mediante paralelismo.

Por ejemplo, si almacenamos una base de datos de tamaño D en un solo disco con rendimiento T el rendimiento del sistema está limitado por T . Por el contrario, si particionamos la base de datos en n discos, cada uno con capacidad D/n y rendimiento T (con suerte equivalente a T), obtenemos un rendimiento ideal de n T que puede ser mejor consumido por múltiples procesadores (idealmente n). Tenga en cuenta que la solución del sistema de base de datos de memoria principal, que intenta mantener la base de datos en memoria principal, es complementaria en lugar de alternativa. En particular, el "cuello de botella de acceso a memoria" en sistemas de memoria principal también puede abordarse utilizando paralelismo de manera similar. Por lo tanto, los diseñadores de sistemas de bases de datos paralelas se han esforzado por desarrollar soluciones orientadas a software para explotar las computadoras paralelas.

Un sistema de bases de datos paralelo puede definirse, en términos generales, como un SGBD implementado en un ordenador paralelo. Esta definición incluye diversas alternativas, desde la adaptación directa de un SGBD existente, que puede requerir únicamente la reescritura de las rutinas de interfaz del sistema operativo, hasta una combinación sofisticada de procesamiento paralelo y funciones del sistema de bases de datos en una nueva arquitectura de hardware/software. Como siempre, existe el equilibrio tradicional entre portabilidad (a varias plataformas) y eficiencia. El enfoque sofisticado permite aprovechar al máximo las oportunidades que ofrece un multiprocesador en detrimento de la portabilidad. Curiosamente, esto ofrece diferentes ventajas a los fabricantes de ordenadores y a los proveedores de software. Por lo tanto, es importante caracterizar los puntos principales en el ámbito de las arquitecturas alternativas de sistemas paralelos. Para ello, precisaremos la solución del sistema de bases de datos paralelo y las funciones necesarias. Esto será útil para comparar las arquitecturas de sistemas de bases de datos paralelos.

Los objetivos de los sistemas de bases de datos paralelas son similares a los de los SGBD distribuidos (rendimiento, disponibilidad, extensibilidad), pero tienen un enfoque ligeramente diferente debido a la mayor interconexión entre los nodos de computación y almacenamiento. Los destacamos a continuación.

1. Alto rendimiento. Esto se puede lograr mediante diversas soluciones complementarias: gestión de datos en paralelo, optimización de consultas y balanceo de carga.

El paralelismo puede utilizarse para aumentar el rendimiento y reducir los tiempos de respuesta de las transacciones. Sin embargo, reducir el tiempo de respuesta de una consulta compleja mediante paralelismo a gran escala puede aumentar su tiempo total (debido a la comunicación adicional) y, como consecuencia, perjudicar el rendimiento. Por lo tanto, es crucial optimizar y paralelizar las consultas para minimizar la sobrecarga del paralelismo, por ejemplo, limitando el grado de paralelismo de la consulta. El balanceo de carga es la capacidad del sistema para distribuir equitativamente una carga de trabajo dada entre todos los procesadores.

Dependiendo de la arquitectura del sistema paralelo, esto se puede lograr de forma estática mediante un diseño de base de datos física adecuado o de forma dinámica en tiempo de ejecución.

2. Alta disponibilidad. Dado que un sistema de base de datos paralela consta de muchos componentes redundantes, puede aumentar considerablemente la disponibilidad de los datos y la tolerancia a fallos. En un sistema altamente paralelo con muchos nodos, la probabilidad de un fallo en un nodo...

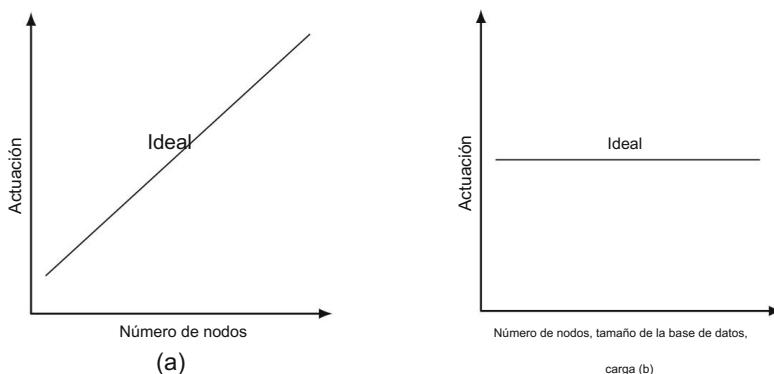


Fig. 8.1 Métricas de extensibilidad. (a) Aceleración lineal. (b) Escalado lineal.

En cualquier momento, el tiempo puede ser relativamente alto. Replicar datos en varios nodos es útil para facilitar la comutación por error, una técnica de tolerancia a fallos que permite la redirección automática de transacciones desde un nodo fallido a otro que almacena una copia de los datos. Esto proporciona un servicio ininterrumpido a los usuarios.

- Extensibilidad. En un sistema paralelo, debería ser más fácil adaptarse al aumento del tamaño de las bases de datos o a las crecientes demandas de rendimiento (p. ej., el rendimiento). La extensibilidad es la capacidad de expandir el sistema sin problemas, añadiendo potencia de procesamiento y almacenamiento. Idealmente, el sistema de bases de datos paralelas debería presentar dos ventajas de extensibilidad: aceleración y escalado lineales (véase la figura 8.1).

La aceleración lineal se refiere a un aumento lineal del rendimiento con un tamaño y una carga de base de datos constantes, mientras que el número de nodos (es decir, la potencia de procesamiento y almacenamiento) aumenta linealmente. El escalamiento lineal se refiere a un rendimiento sostenido con un aumento lineal tanto del tamaño de la base de datos, como de la carga y el número de nodos. Además, la ampliación del sistema debería requerir una reorganización mínima de la base de datos existente.

El creciente uso de clústeres en aplicaciones a gran escala, como la gestión de datos web, ha dado lugar al uso del término "escalamiento horizontal" en lugar de "escalamiento vertical". La Figura 8.2 muestra un clúster con cuatro servidores, cada uno con un número determinado de nodos de procesamiento ("P"). En este contexto, el escalamiento vertical (también llamado "escalamiento vertical") se refiere a la adición de más nodos a un servidor, lo que limita su tamaño máximo. El escalamiento horizontal (también llamado "escalamiento horizontal") se refiere a la adición de más servidores, denominados "servidores de escalamiento horizontal", de forma flexible, para escalar casi infinitamente.

8.2 Arquitecturas paralelas

Un sistema de base de datos paralela representa un compromiso en las decisiones de diseño para ofrecer las ventajas mencionadas con una buena relación calidad-precio. Una decisión de diseño clave es la forma en que se integran los principales elementos de hardware, es decir, los procesadores.

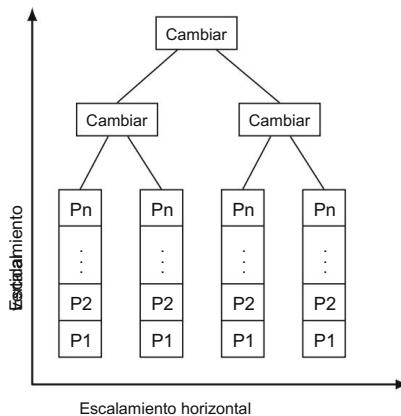


Figura 8.2 Escalamiento vertical versus escalamiento horizontal

La memoria principal y los discos están conectados a través de alguna red de interconexión.

En esta sección, presentamos los aspectos arquitectónicos de los sistemas de bases de datos paralelos.

En particular, presentamos y comparamos las tres arquitecturas paralelas básicas: memoria compartida, disco compartido y sin recursos compartidos. La memoria compartida se utiliza en multiprocesadores estrechamente acoplados, mientras que la sin recursos compartidos y el disco compartido se utilizan en clústeres.

Al describir estas arquitecturas, nos centramos en los cuatro elementos principales de hardware: interconexión, procesadores (P), módulos de memoria principal (M) y discos. Para simplificar, ignoramos otros elementos como la caché del procesador, los núcleos del procesador y el bus de E/S.

8.2.1 Arquitectura general

Partiendo de una arquitectura cliente-servidor, las funciones que soporta un sistema de base de datos paralelo pueden dividirse en tres subsistemas, de forma similar a un SGBD típico. Sin embargo, las diferencias radican en la implementación de estas funciones, que ahora deben gestionar el paralelismo, la partición y replicación de datos, y las transacciones distribuidas. Dependiendo de la arquitectura, un nodo de procesador puede soportar todos (o un subconjunto) de estos subsistemas. La Figura 8.3 muestra la arquitectura que utiliza estos subsistemas, basada en la arquitectura de la Figura 1.11 con la incorporación de un gestor de clientes.

1. Administrador de clientes. Proporciona soporte para las interacciones de los clientes con el sistema de base de datos paralela. En particular, gestiona las conexiones y desconexiones entre los procesos de cliente, que se ejecutan en diferentes servidores, por ejemplo, servidores de aplicaciones, y los procesadores de consultas. Por lo tanto, inicia consultas de cliente (que pueden ser transacciones) en algunos procesadores de consultas, que a su vez se encargan de interactuar directamente con los clientes y de procesar las consultas y gestionar las transacciones. El administrador de clientes también realiza el balanceo de carga mediante

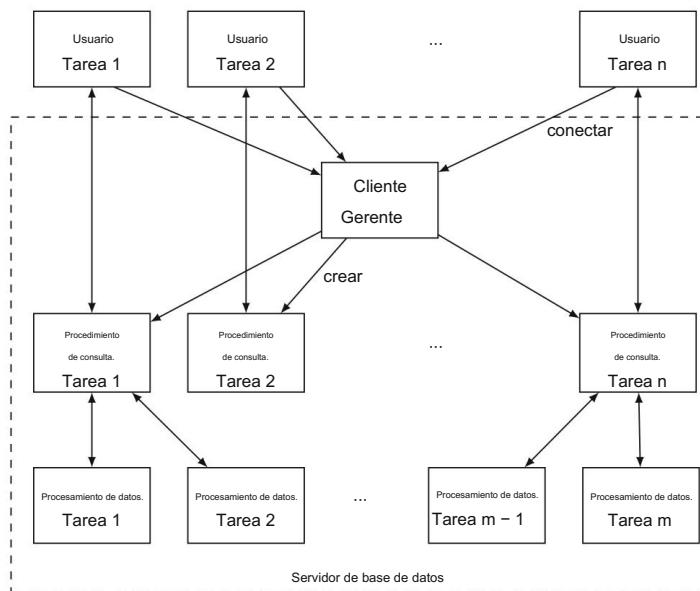


Fig. 8.3 Arquitectura general de un sistema de base de datos paralela

Un catálogo que mantiene información sobre la carga de los nodos del procesador y las consultas precompiladas (incluida la ubicación de los datos). Esto permite activar la ejecución de consultas precompiladas en procesadores de consultas ubicados cerca de los datos a los que se accede.

El administrador de clientes es un proceso ligero y, por lo tanto, no supone un cuello de botella. Sin embargo, para garantizar la tolerancia a fallos, puede replicarse en varios nodos.

2. Procesador de consultas. Recibe y gestiona las consultas del cliente, como compilar consultas, ejecutar consultas e iniciar transacciones. Utiliza el directorio de la base de datos, que contiene toda la metainformación sobre datos, consultas y transacciones. El directorio debe gestionarse como una base de datos, replicable en todos los nodos del procesador de consultas.
- Según la solicitud, activa las distintas fases de compilación, incluyendo el control semántico de datos, la optimización y paralelización de consultas; activa y supervisa la ejecución de consultas mediante los procesadores de datos; y devuelve los resultados, así como los códigos de error, al cliente. También puede activar la validación de transacciones en los procesadores de datos.
3. Procesador de datos. Gestiona los datos de la base de datos y del sistema (registro del sistema, etc.) y proporciona todas las funciones básicas necesarias para ejecutar consultas en paralelo, como la ejecución de operadores de base de datos, el soporte para transacciones paralelas, la gestión de caché, etc.

8.2.2 Memoria compartida

En el enfoque de memoria compartida, cualquier procesador tiene acceso a cualquier módulo de memoria o unidad de disco mediante una interconexión. Todos los procesadores están bajo el control de un único sistema operativo.

Una ventaja importante es la simplicidad del modelo de programación basado en memoria virtual compartida. Dado que la metainformación (directorio) y la información de control (p. ej., tablas de bloqueo) pueden ser compartidas por todos los procesadores, la creación de software de bases de datos no difiere mucho de la de computadoras con un solo procesador. En particular, el paralelismo entre consultas es gratuito. El paralelismo intraconsultas requiere cierta parallelización, pero sigue siendo bastante simple. El balanceo de carga también es sencillo, ya que puede lograrse en tiempo de ejecución utilizando la memoria compartida, asignando cada nueva tarea al procesador con menos actividad.

Dependiendo de si se comparte la memoria física, son posibles dos enfoques: Acceso Uniforme a Memoria (UMA) y Acceso No Uniforme a Memoria (NUMA), que presentamos a continuación.

8.2.2.1 Acceso uniforme a memoria (UMA)

Con UMA, la memoria física es compartida por todos los procesadores, por lo que el acceso a la memoria es constante (véase la Fig. 8.4). Por ello, también se le denomina multiprocesador simétrico (SMP). Las topologías de red comunes para interconectar procesadores incluyen bus, barra cruzada y malla.

Las primeras SMP aparecieron en la década de 1960 para mainframes y contaban con pocos procesadores. En la década de 1980, aparecieron máquinas SMP más grandes, con decenas de procesadores. Sin embargo, sufrían de un alto coste y una escalabilidad limitada. La interconexión, que requiere hardware bastante complejo debido a la necesidad de conectar cada procesador a cada módulo o disco de memoria, supuso un alto coste. Con procesadores cada vez más rápidos (incluso con cachés más grandes), los accesos conflictivos a la memoria compartida...

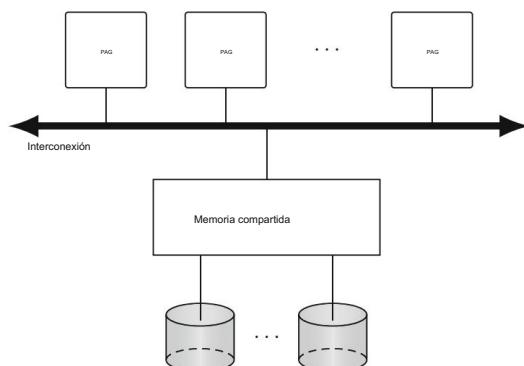


Fig. 8.4 Memoria compartida

Aumentan rápidamente y degradan el rendimiento. Por lo tanto, la escalabilidad se ha limitado a menos de diez procesadores. Finalmente, dado que el espacio de memoria es compartido por todos los procesadores, un fallo de memoria puede afectar a la mayoría de ellos, lo que perjudica la disponibilidad de los datos.

Los procesadores multinúcleo también se basan en SMP, con múltiples núcleos de procesamiento y memoria compartida en un solo chip. En comparación con los diseños SMP multichip anteriores, mejoran el rendimiento de las operaciones de caché, requieren mucho menos espacio en la placa de circuito impreso y consumen menos energía. Por lo tanto, la tendencia actual en el desarrollo de procesadores multinúcleo apunta a un número cada vez mayor de núcleos, a medida que se hacen viables los procesadores con cientos de núcleos.

Algunos ejemplos de sistemas de bases de datos paralelas SMP incluyen XPRS, DBS3 y Volcano.

8.2.2.2 Acceso a memoria no uniforme (NUMA)

El objetivo de NUMA es proporcionar un modelo de programación de memoria compartida y todas sus ventajas, en una arquitectura escalable con memoria distribuida. Cada procesador tiene su propio módulo de memoria local, al que puede acceder eficientemente. El término NUMA refleja que los accesos a la memoria (virtualmente) compartida tienen un coste diferente según si la memoria física es local o remota al procesador.

La clase más antigua de sistemas NUMA son los multiprocesadores NUMA con coherencia de caché (CC-NUMA) (véase la Fig. 8.5). Dado que diferentes procesadores pueden acceder a los mismos datos en un modo de actualización conflictivo, se requieren protocolos globales de consistencia de caché. Para que el acceso remoto a memoria sea eficiente, una solución consiste en que la consistencia de caché se realice en hardware mediante una interconexión de caché consistente especial. Dado que el hardware admite la memoria compartida y la consistencia de caché, el acceso remoto a memoria es muy eficiente, con un coste solo varias veces superior (normalmente hasta tres veces superior) al del acceso local.

Un enfoque más reciente para NUMA consiste en aprovechar la capacidad de acceso directo a memoria remota (RDMA), que ahora ofrecen las interconexiones de clúster de baja latencia, como Infiniband y Myrinet. RDMA se implementa en el hardware de la tarjeta de red y proporciona una red de copia cero, lo que permite que un nodo del clúster acceda directamente a la memoria de otro nodo sin necesidad de copiar entre los búferes del sistema operativo. Esto produce un acceso a memoria remota típico con latencias del orden de 10 veces superiores a las del acceso a memoria local. Sin embargo, aún hay margen de mejora.

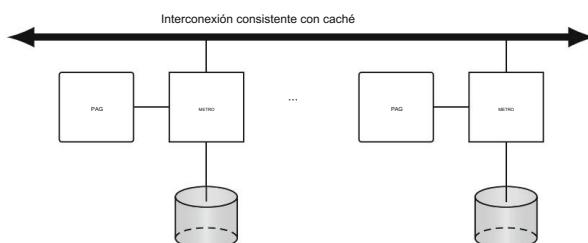


Fig. 8.5 Arquitectura de memoria caché coherente no uniforme (CC-NUMA)

Por ejemplo, la integración más estrecha del control de memoria remota en la jerarquía de coherencia local del nodo permite el acceso remoto con latencias cuatro veces superiores a las del acceso local. Por lo tanto, RDMA puede aprovecharse para mejorar el rendimiento de las operaciones paralelas de bases de datos. Sin embargo, requiere nuevos algoritmos compatibles con NUMA para abordar el cuello de botella del acceso a memoria remota. El enfoque básico consiste en maximizar el acceso a memoria local mediante una planificación cuidadosa de las tareas del SGBD cercanas a los datos e intercalar la computación y la comunicación de red.

Los multiprocesadores modernos utilizan una arquitectura jerárquica que mezcla NUMA y UMA, es decir, un multiprocesador NUMA donde cada procesador es un procesador multinúcleo.

A su vez, cada multiprocesador NUMA puede utilizarse como un nodo en un clúster.

8.2.3 Disco compartido

En un clúster de discos compartidos (véase la Fig. 8.6), cualquier procesador tiene acceso a cualquier unidad de disco a través de la interconexión, pero acceso exclusivo (no compartido) a su memoria principal. Cada nodo de procesador-memoria, que puede ser un nodo de memoria compartida, está bajo el control de su propia copia del sistema operativo. Por lo tanto, cada procesador puede acceder a las páginas de la base de datos en el disco compartido y almacenarlas en caché en su propia memoria. Dado que diferentes procesadores pueden acceder a la misma página en modos de actualización conflictivos, se requiere consistencia global de la caché. Esto se logra típicamente mediante un gestor de bloqueos distribuido que se puede implementar mediante las técnicas descritas en el Cap. 5. El primer DBMS paralelo que utilizó discos compartidos fue Oracle, con una implementación eficiente de un gestor de bloqueos distribuido para la consistencia de la caché. Ha evolucionado hasta la máquina de base de datos Oracle Exadata. Otros proveedores importantes de DBMS, como IBM, Microsoft y Sybase, también ofrecen implementaciones de discos compartidos, generalmente para cargas de trabajo OLTP.

El disco compartido requiere que los nodos del clúster puedan acceder globalmente a los discos.

Existen dos tecnologías principales para compartir discos en un clúster: almacenamiento conectado a red (NAS) y red de área de almacenamiento (SAN). Un NAS es un dispositivo dedicado a compartir discos a través de una red (generalmente TCP/IP) que utiliza un protocolo de sistema de archivos distribuido como el Sistema de Archivos de Red (NFS). El NAS es ideal para aplicaciones de bajo rendimiento, como la copia de seguridad y el archivado de datos desde discos duros de PC. Sin embargo, es relativamente lento y no es adecuado para la gestión de bases de datos, ya que se ejecuta rápidamente.

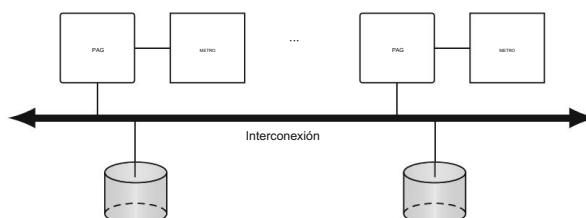


Fig. 8.6 Arquitectura de disco compartido

Se convierte en un cuello de botella con muchos nodos. Una red de área de almacenamiento (SAN) ofrece una funcionalidad similar, pero con una interfaz de nivel inferior. Para mayor eficiencia, utiliza un protocolo basado en bloques, lo que facilita la gestión de la consistencia de la caché (a nivel de bloque). Como resultado, la SAN proporciona un alto rendimiento de datos y puede escalar a un gran número de nodos.

El disco compartido ofrece tres ventajas principales: administración sencilla y económica, alta disponibilidad y buen equilibrio de carga. Los administradores de bases de datos no necesitan lidiar con particiones complejas de datos, y el fallo de un nodo solo afecta a sus datos en caché, mientras que los datos en disco siguen disponibles para los demás nodos. Además, el equilibrio de carga es sencillo, ya que cualquier solicitud puede ser procesada por cualquier nodo de procesador-memoria.

Las principales desventajas son el coste (debido a la SAN) y la escalabilidad limitada, causada por el posible cuello de botella y la sobrecarga de los protocolos de coherencia de caché para bases de datos muy grandes. Una solución es recurrir al particionamiento de datos, como en el modelo de no compartido, a costa de una administración más compleja.

8.2.4 Nada compartido

En un clúster sin recursos compartidos (ver Figura 8.7), cada procesador tiene acceso exclusivo a su memoria principal y a su disco, mediante almacenamiento conectado directamente (DAS).

Cada nodo de procesador, memoria y disco está bajo el control de su propia copia del sistema operativo. Los clústeres sin recursos compartidos se utilizan ampliamente en la práctica, generalmente con nodos NUMA, ya que ofrecen la mejor relación calidad-precio y escalan a configuraciones muy grandes (miles de nodos).

Cada nodo puede considerarse un sitio local (con su propia base de datos y software) en un SGBD distribuido. Por lo tanto, la mayoría de las soluciones diseñadas para estos sistemas, como la fragmentación de bases de datos, la gestión distribuida de transacciones y el procesamiento distribuido de consultas, pueden reutilizarse. Mediante una interconexión rápida, es posible alojar un gran número de nodos. A diferencia del SMP, esta arquitectura suele denominarse Procesador Masivamente Paralelo (MPP).

Al favorecer el crecimiento gradual y fluido del sistema mediante la adición de nuevos nodos, el modelo de no compartido proporciona extensibilidad y escalabilidad. Sin embargo, requiere...

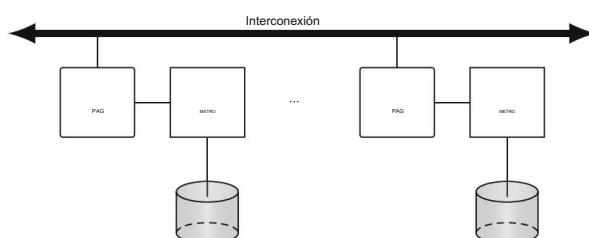


Fig. 8.7 Arquitectura de no compartir nada

Partitionar cuidadosamente los datos en varios discos. Además, la adición de nuevos nodos al sistema probablemente requiera reorganizar y reparticionar la base de datos para solucionar los problemas de equilibrio de carga. Finalmente, la tolerancia a fallos de los nodos es difícil (requiere replicación), ya que un nodo fallido dejará sus datos en disco indisponibles.

Muchos prototipos de sistemas de bases de datos paralelas han adoptado la arquitectura de no compartido, como Bubba, Gamma, Grace y Prisma/DB. El primer producto importante de SGBD paralelo fue la máquina de base de datos de Teradata. Otras empresas importantes de SGBD, como IBM, Microsoft y Sybase, y proveedores de SGBD con almacenamiento en columnas, como MonetDB y Vertica, ofrecen implementaciones de no compartido para aplicaciones OLAP de alta gama. Por último, los SGBD NoSQL y los sistemas de big data suelen utilizar la arquitectura de no compartido.

Tenga en cuenta que es posible tener una arquitectura híbrida, donde una parte del clúster no comparte nada, por ejemplo, para cargas de trabajo OLAP, y otra parte comparte discos, por ejemplo, para cargas de trabajo OLTP. Por ejemplo, Teradata admite el concepto de camarilla (un conjunto de nodos que comparten un conjunto común de discos) en su arquitectura de no compartición para mejorar la disponibilidad.

8.3 Ubicación de datos

En el resto de este capítulo, consideraremos una arquitectura de no partición, ya que es el caso más general y sus técnicas de implementación también se aplican, a veces de forma simplificada, a otras arquitecturas. La ubicación de datos en un sistema de bases de datos paralelas presenta similitudes con la fragmentación de datos en bases de datos distribuidas (véase el capítulo 2). Una similitud obvia es que la fragmentación puede utilizarse para aumentar el paralelismo. Como se mencionó en el capítulo 2, los SGBD paralelos utilizan principalmente particionamiento horizontal, aunque la fragmentación vertical también puede utilizarse para aumentar el paralelismo y el equilibrio de carga, al igual que en las bases de datos distribuidas, y se ha empleado en SGBD con almacenamiento en columnas, como MonetDB o Vertica. Otra similitud con las bases de datos distribuidas es que, dado que los datos son mucho más grandes que los programas, la ejecución debe ocurrir, en la medida de lo posible, donde residen los datos. Como se mencionó en el capítulo 2, existen dos diferencias importantes con el enfoque de bases de datos distribuidas. En primer lugar, no es necesario maximizar el procesamiento local (en cada nodo), ya que los usuarios no están asociados a nodos específicos. En segundo lugar, el equilibrio de carga es mucho más difícil de lograr con un gran número de nodos. El principal problema es evitar la contención de recursos, que puede provocar la sobrecarga de todo el sistema (por ejemplo, un nodo termina haciendo todo el trabajo, mientras que los demás permanecen inactivos). Dado que los programas se ejecutan donde residen los datos, la ubicación de estos es crucial para el rendimiento.

Las estrategias de particionamiento de datos más comunes que se utilizan en los SGBD paralelos son los enfoques round-robin, hash y particionamiento por rango, que se describen en la Sección 2.1.1. El particionamiento de datos debe escalar con el aumento del tamaño y la carga de la base de datos. Por lo tanto, el grado de particionamiento, es decir, el número de nodos en los que se partitiona una relación, debe ser una función del tamaño y la frecuencia de acceso de la relación. Por lo tanto, aumentar el grado de particionamiento puede resultar en la ubicación

Reorganización. Por ejemplo, una relación inicialmente distribuida en ocho nodos puede duplicar su cardinalidad mediante inserciones posteriores, en cuyo caso debería distribuirse en 16 nodos.

En un sistema altamente paralelo con particionamiento de datos, las reorganizaciones periódicas para equilibrar la carga son esenciales y deben ser frecuentes, a menos que la carga de trabajo sea bastante estática y solo experimente algunas actualizaciones. Dichas reorganizaciones deben ser transparentes para las consultas compiladas que se ejecutan en el servidor de bases de datos. En particular, las consultas no deben recompilarse debido a la reorganización y deben ser independientes de la ubicación de los datos, que puede cambiar rápidamente. Esta independencia se puede lograr si el sistema en tiempo de ejecución admite el acceso asociativo a los datos distribuidos. Esto difiere de un SGBD distribuido, donde el acceso asociativo se logra en tiempo de compilación mediante el procesador de consultas utilizando el directorio de datos.

Una solución al acceso asociativo es replicar un mecanismo de índice global en cada nodo. El índice global indica la ubicación de una relación en un conjunto de nodos. Conceptualmente, el índice global es un índice de dos niveles con una agrupación principal en el nombre de la relación y una agrupación secundaria en algún atributo de la relación.

Este índice global admite la partición de variables, donde cada relación tiene un grado de partición diferente. La estructura del índice puede basarse en hash o en una organización tipo árbol B. En ambos casos, las consultas de coincidencia exacta se pueden procesar eficientemente con un solo acceso a un nodo. Sin embargo, con el hash, las consultas de rango se procesan accediendo a todos los nodos que contienen datos de la relación consultada. El uso de un índice de árbol B (generalmente mucho más grande que un índice hash) permite un procesamiento más eficiente de las consultas de rango, donde solo se accede a los nodos que contienen datos en el rango especificado.

Ejemplo 8.1 La Figura 8.8 proporciona un ejemplo de un índice global y un índice local para la relación EMP(ENO, ENAME, TITLE) del ejemplo de base de datos de ingeniería que hemos estado utilizando en este libro.

Supongamos que queremos localizar los elementos en relación con EMP con el valor ENO “E50”. El índice de primer nivel asigna el nombre EMP al índice del atributo ENO.

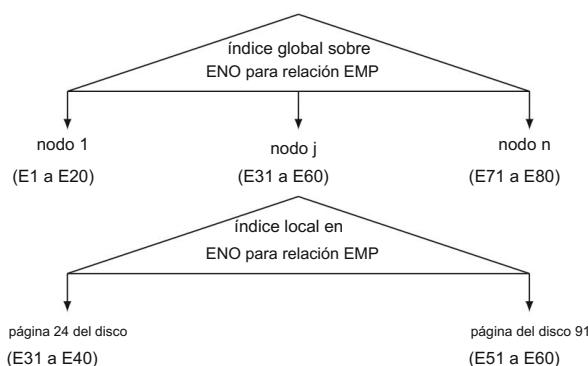


Fig. 8.8 Ejemplo de índices globales y locales

Para la relación EMP. Luego, el índice de segundo nivel asigna el valor del clúster «E50» al nodo número j. También se requiere un índice local dentro de cada nodo para mapear una relación a un conjunto de páginas de disco dentro del nodo. El índice local tiene dos niveles: una agrupación principal por nombre de relación y una agrupación secundaria por atributo. El atributo de agrupamiento menor del índice local es el mismo que el del índice global. Por lo tanto, se mejora el enrutamiento asociativo de un nodo a otro según (nombre de la relación, valor del clúster). Este índice local asigna además el valor del clúster "E5" a la página número 91.

Un problema grave en la ubicación de datos es lidiar con distribuciones de datos sesgadas que pueden provocar particiones no uniformes y perjudicar el equilibrio de carga. Una solución es tratar las particiones no uniformes adecuadamente, por ejemplo, fragmentando aún más las particiones grandes. Esto es fácil con la partición por rangos, ya que una partición puede dividirse como una hoja de árbol B, con cierta reorganización del índice local. Con el hash, la solución es usar una función hash diferente para cada atributo. La separación entre nodos lógicos y físicos es útil en este caso, ya que un nodo lógico puede corresponder a varios nodos físicos.

Un último factor que complica la ubicación de los datos es la replicación para lograr alta disponibilidad, que analizamos en detalle en el capítulo 6. En sistemas de gestión de bases de datos (SGBD) paralelos, se pueden adoptar enfoques más sencillos, como la arquitectura de discos en espejo , donde se mantienen dos copias de los mismos datos: una principal y una de respaldo. Sin embargo, en caso de fallo de un nodo, la carga del nodo con la copia puede duplicarse, lo que perjudica el equilibrio de carga. Para evitar este problema, se han propuesto diversas estrategias de replicación de datos de alta disponibilidad para sistemas de bases de datos paralelos. Una solución interesante es el particionamiento intercalado de Teradata, que divide la copia de respaldo en varios nodos. La figura 8.9 ilustra el particionamiento intercalado de la relación R en cuatro nodos, donde cada copia principal de una partición, por ejemplo, R1, se divide a su vez en tres particiones, por ejemplo, R1,1, R1,2 y R1,3, cada una en un nodo de respaldo diferente. En caso de fallo, la carga de la copia principal se equilibra entre los nodos de respaldo. Pero si dos nodos fallan, no se puede acceder a la relación, lo que afecta la disponibilidad. Reconstruir la copia principal a partir de sus copias de seguridad independientes puede ser costoso. En modo normal, mantener la consistencia de la copia también puede ser costoso.

Una solución alternativa es la partición encadenada de Gamma . que almacena la copia principal y la de respaldo en dos nodos adyacentes (Fig. 8.10). La idea principal es que

Node	1	2	3	4
Primary copy	R ₁	R ₂	R ₃	R ₄
Backup copies		R _{1,1}	R _{1,2}	R _{1,3}
	R _{2,1}		R _{2,2}	R _{2,3}
	R _{3,1}	R _{3,2}		R _{3,3}
	R _{4,1}	R _{4,2}	R _{4,3}	

Fig. 8.9 Ejemplo de partición intercalada

Node	1	2	3	4
Primary copy	R ₁	R ₂	R ₃	R ₄
Backup copy	R ₄	R ₁	R ₂	R ₃

Fig. 8.10 Ejemplo de partición encadenada

La probabilidad de que dos nodos adyacentes fallen es mucho menor que la de que dos nodos cualesquiera fallen. En modo de fallo, la carga del nodo fallido y de los nodos de respaldo se equilibra entre los nodos restantes mediante el uso de nodos de copia de seguridad y nodos principales. Además, mantener la consistencia de la copia es más económico. Un problema pendiente es cómo realizar la ubicación de los datos teniendo en cuenta la replicación. Al igual que la asignación de fragmentos en bases de datos distribuidas, esto debería considerarse un problema de optimización.

8.4 Procesamiento de consultas paralelas

El objetivo del procesamiento de consultas paralelas es transformar las consultas en planes de ejecución que puedan ejecutarse eficientemente en paralelo. Esto se logra aprovechando la ubicación paralela de datos y las diversas formas de paralelismo que ofrecen las consultas de alto nivel. En esta sección, primero presentamos los algoritmos paralelos básicos para el procesamiento de datos. A continuación, analizamos la optimización de consultas paralelas.

8.4.1 Algoritmos paralelos para el procesamiento de datos

La ubicación de datos particionados es la base para la ejecución paralela de consultas de bases de datos. Dada la ubicación de datos particionados, un aspecto importante es el diseño de algoritmos paralelos para el procesamiento eficiente de operadores de bases de datos (es decir, operadores de álgebra relacional) y consultas de bases de datos que combinan múltiples operadores. Este aspecto es complejo porque se debe lograr un buen equilibrio entre paralelismo y coste de comunicación, ya que un mayor paralelismo implica mayor comunicación entre procesadores.

Los algoritmos paralelos para operadores de álgebra relacional son los componentes básicos necesarios para el procesamiento paralelo de consultas. El objetivo de estos algoritmos es maximizar el grado de paralelismo. Sin embargo, según la ley de Amdahl, solo una parte de un algoritmo puede paralelizarse. Sea seq la razón de la parte secuencial de un programa (un valor entre 0 y 1), es decir, la que no puede paralelizarse, y sea p el número de procesadores. La aceleración máxima que se puede alcanzar viene dada por la siguiente fórmula:

$$\text{MaxSpeedup}(\text{secuencia}, p) = \frac{1}{\text{secuencia} + \frac{1-\text{secuencia}}{p}}$$

Por ejemplo, con $\text{seq} = 0$ (todo el programa es paralelo) y $p = 4$, obtenemos la aceleración ideal, es decir, 4. Pero con $\text{seq} = 0,3$, la aceleración disminuye a 2,1. Incluso si duplicamos el número de procesadores, es decir, $p = 8$, la aceleración solo aumenta ligeramente a 2,5. Por lo tanto, al diseñar algoritmos paralelos para el procesamiento de datos, es importante minimizar la parte secuencial de un algoritmo y maximizar la parte paralela, aprovechando el paralelismo intraoperador.

El procesamiento del operador SELECT en un contexto de ubicación de datos particionados es idéntico al de una base de datos distribuida fragmentada. Dependiendo del predicado SELECT, el operador puede ejecutarse en un solo nodo (en el caso de un predicado de coincidencia exacta) o, en el caso de predicados de complejidad arbitraria, en todos los nodos sobre los que se partitiona la relación. Si el índice global se organiza con una estructura similar a un árbol B (véase la figura 8.8), un operador SELECT con un predicado de rango solo puede ser ejecutado por los nodos que almacenan datos relevantes. En el resto de esta sección, nos centraremos en el procesamiento paralelo de los dos operadores principales utilizados en las consultas de bases de datos: sort y join.

8.4.1.1 Algoritmos de ordenamiento paralelo

Las relaciones de ordenación son necesarias para consultas que requieren un resultado ordenado o implican agregación y agrupación. Además, es difícil hacerlo eficientemente, ya que cada elemento debe compararse con los demás. Uno de los algoritmos de ordenación de un solo procesador más rápidos es quicksort, pero es altamente secuencial y, por lo tanto, según la ley de Amdahl, no es adecuado para la adaptación en paralelo. Varios otros algoritmos de ordenación centralizados pueden implementarse en paralelo. Uno de los algoritmos más populares es el algoritmo de ordenación por fusión paralela, debido a su fácil implementación y a que no presenta requisitos estrictos en la arquitectura del sistema paralelo. Por lo tanto, se ha utilizado tanto en clústeres de disco compartido como en clústeres sin recursos compartidos. También puede adaptarse para aprovechar las ventajas de los procesadores multinúcleo.

Revisamos brevemente el algoritmo de ordenamiento por fusión de b-vías. Consideremos un conjunto de n elementos a ordenar. Una serie se define como una secuencia ordenada de elementos; por lo tanto, el conjunto a ordenar contiene n series de un elemento. El método consiste en la fusión iterativa de b series de K elementos en una serie ordenada de K/b elementos, comenzando con $K = 1$. Para el paso i , cada conjunto de b series de b^{i-1} elementos se fusiona en una serie ordenada de b^i elementos. A partir de $i = 1$, el número de pasos necesarios para ordenar n elementos es $\log_b n$.

A continuación, describimos la aplicación de este método en un clúster sin recursos compartidos. Utilizamos el popular modelo maestro-trabajador para ejecutar tareas paralelas, donde un nodo maestro coordina las actividades de los nodos trabajadores, enviándoles tareas y datos, y recibiendo notificaciones de las tareas realizadas.

Supongamos que tenemos que ordenar una relación de p páginas de disco particionadas en n nodos. Cada nodo tiene una memoria local de $b+1$ páginas, donde b páginas se usan como páginas de entrada y 1 como página de salida. El algoritmo procede en dos etapas. En la primera etapa, cada nodo ordena localmente su fragmento, p. ej., usando quicksort si el nodo es un solo procesador o un ordenamiento por fusión en paralelo b-way si el nodo es un procesador multinúcleo. Esta etapa se llama etapa óptima ya que todos los nodos están completamente ocupados. Genera n ejecuciones de p/n páginas, y si n es igual a b , un nodo puede fusionarlas en una sola pasada. Sin embargo, n puede ser mucho mayor que b , en cuyo caso la solución es que el nodo maestro ordene los nodos de trabajo como un trie de orden b durante la última etapa, llamada etapa postóptima. El número de nodos necesarios se divide por b en cada pasada.

En la última pasada, un nodo fusiona toda la relación. El número de pasadas para la etapa posóptima es de $\log bp$. Esta etapa degrada el grado de paralelismo.

8.4.1.2 Algoritmos de unión paralela

Suponiendo dos relaciones particionadas arbitrarias, existen tres algoritmos paralelos básicos para unirlas: el algoritmo de combinación de ordenamiento por fusión paralela, el algoritmo de bucles anidados paralelos (PNL) y el algoritmo de combinación hash paralela (PHJ). Estos algoritmos son variaciones de su contraparte centralizada. El algoritmo de combinación de ordenamiento por fusión paralela simplemente ordena ambas relaciones según el atributo de unión mediante un ordenamiento por fusión paralela y las une mediante una operación similar a la de una fusión realizada por un solo nodo. Aunque la última operación es secuencial, la relación unida resultante se ordena según el atributo de unión, lo cual puede ser útil para la siguiente operación.

Los otros dos algoritmos son completamente paralelos. Los describimos con más detalle utilizando un lenguaje de programación pseudoconcurrente con tres construcciones principales: parallel-do, send y receive. Parallel-do especifica que el siguiente bloque de acciones se ejecuta en paralelo. Por ejemplo:

```
para i de 1 a n en paralelo-hacer la acción A
```

Indica que la acción A será ejecutada por n nodos en paralelo. Las construcciones de envío y recepción son primitivas básicas de comunicación de datos: el envío envía datos de un nodo a uno o más nodos, mientras que la recepción obtiene el contenido de los datos enviados a un nodo específico. A continuación, consideraremos la unión de dos relaciones R y S , divididas en m y n nodos, respectivamente. Para simplificar, asumimos que los m nodos son distintos de los n nodos. Un nodo que contiene un fragmento de R (respectivamente, S) se denomina nodo R (respectivamente, nodo S).

Algoritmo de unión de bucles anidados paralelos

El algoritmo de bucles anidados paralelos es simple y general. Implementa el método de fragmentación y replicación descrito en la Sección 4.5.1. Básicamente, se compone de:

Algoritmo 8.1: Bucle anidado paralelo (PNL)

Entrada: R1, R2,..., Rm: fragmentos de la relación R S1,

S2,..., Sn: fragmentos de la relación S ; JP:

predicado de unión

Salida: T1, T2,..., Tn: fragmentos de resultado

```

comienzan para i de 1 a m en paralelo hacen {envían R completamente a cada nodo S} envían Ri a cada
|   nodo que contiene un fragmento de S terminan para para
j de 1 a
n en paralelo hacen i=1 Ri; Tj ← R           {realizar la unión en cada nodo S}
|   R ←   JP Sj                                {Ri de los nodos R; R se replica completamente en los nodos S}
|   terminan para
fin

```

Producto cartesiano de las relaciones R y S en paralelo. Por lo tanto, se pueden admitir predicados de unión de complejidad arbitraria, no solo de igualdad.

El algoritmo ejecuta dos bucles anidados. Una relación se elige como la relación interna, a la que se accede en el bucle interno, y la otra como la relación externa, a la que se accede en el bucle externo. Esta elección depende de una función de coste con dos parámetros principales: el tamaño de las relaciones, que afecta al coste de la comunicación, y la presencia de índices en los atributos de unión, que afecta al coste del procesamiento de la unión local.

Este algoritmo se describe en el Algoritmo 8.1, donde el resultado de la unión se genera en los nodos S, es decir, se elige S como relación interna. El algoritmo se desarrolla en dos fases.

En la primera fase, cada fragmento de R se envía y replica en cada nodo que contiene un fragmento de S (existen n nodos de este tipo). Esta fase se realiza en paralelo por m nodos; por lo tanto, se requieren (m × n) mensajes.

En la segunda fase, cada nodo S j recibe la relación R en su totalidad y une localmente R con el fragmento Sj . Esta fase se realiza en paralelo por n nodos. La unión local puede realizarse como en un SGBD centralizado. Dependiendo del algoritmo de unión local, el procesamiento de la unión puede comenzar o no en cuanto se reciben los datos. Si se utiliza una unión de bucle anidado, el algoritmo, posiblemente con un índice en el atributo de unión de S, , el procesamiento de la unión... se puede ejecutar de forma segmentada tan pronto como llega una tupla de R. Si, por otro lado, se utiliza un algoritmo de unión de ordenación y fusión, todos los datos deben haberse recibido antes de que comience la unión de las relaciones ordenadas.

En resumen, el algoritmo de bucle anidado paralelo puede considerarse como un reemplazo del operador RS por ni=1(R Si).

Ejemplo 8.2 La figura 8.11 muestra la aplicación del algoritmo de bucle anidado paralelo con m = n = 2.

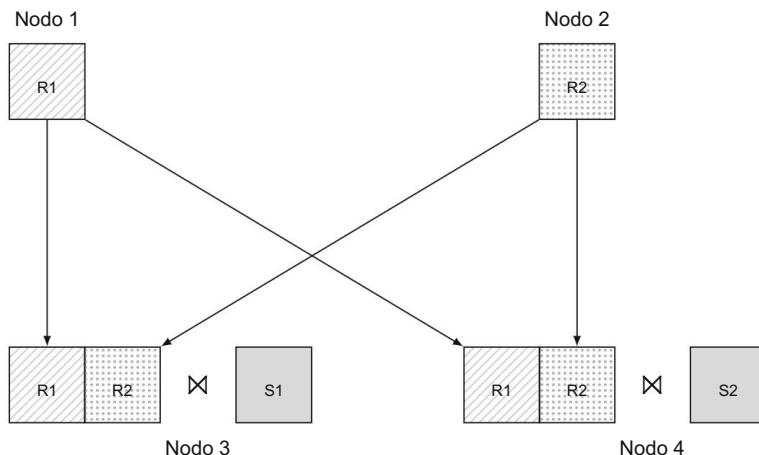


Fig. 8.11 Ejemplo de bucle anidado paralelo

Algoritmo de unión hash paralela

El algoritmo de unión hash paralela que se muestra en el algoritmo 8.2 se aplica solo en el caso de equijoin y no requiere ninguna partición particular de las relaciones de operandos.

Se propuso por primera vez para la máquina de base de datos Grace y se conoce como Grace unión hash.

La idea básica es dividir las relaciones R y S en el mismo número p de conjuntos mutuamente excluyentes (fragmentos) R_1, R_2, \dots, R_p , y S_1, S_2, \dots, S_p , tales que

$$RS = \sum_{i=1}^p (R_i S_i)$$

La partición de R y S se basa en la misma función hash aplicada a la Atributo de unión. Cada unión individual ($R_i S_i$) se realiza en paralelo y el resultado de la unión se produce en p nodos. Estos p nodos pueden seleccionarse en tiempo de ejecución según la carga del sistema.

El algoritmo se puede dividir en dos fases principales, una fase de construcción y una fase de prueba. La fase de construcción genera un hash de R usado como relación interna, en el atributo de unión, envía a los nodos de destino p que construyen una tabla hash para las tuplas entrantes. La sonda La fase envía S, la relación externa, asociativamente a los nodos p de destino que sondean la Tabla hash para cada tupla entrante. Por lo tanto, una vez creadas las tablas hash Para R, las tuplas S se pueden enviar y procesar en una canalización sondeando las tablas hash.

Ejemplo 8.3 La figura 8.12 muestra la aplicación del algoritmo de unión hash paralela con $m = n = 2$. Suponemos que el resultado se produce en los nodos 1 y 2. Por lo tanto, Una flecha del nodo 1 al nodo 1 o del nodo 2 al nodo 2 indica una transferencia local.

Algoritmo 8.2: Unión Hash Paralela (PHJ)

Entrada: R1, R2,..., Rm: fragmentos de la relación R;
 S1, S2,..., Sn: fragmentos de la relación S;
 JP: predicado de unión RA = SB;
 h: función hash que devuelve un elemento de [1, p]
 Salida: T1, T2,..., Tp: fragmentos de resultados

comenzar

{Fase de construcción}

para i de 1 a m en paralelo hacer

$R_{j_i} \leftarrow$ aplicar $h(A)$ a R_i ($j = 1, \dots, p$); enviar R_j
 i al nodo j

fin para

para j de 1 a p en paralelo hacer

$R_j \leftarrow \bigcup_{i=1}^m R_{j_i}$ {Recibir fragmentos R_j de los nodos R}
 crear una tabla hash local para R_j

fin para

{Fase de sonda}

para i de 1 a n en paralelo hacer

$S_{j_i} \leftarrow$ aplicar $h(B)$ a S_i ($j = 1, \dots, p$); enviar S_j
 i al nodo j

fin para

para j de 1 a p en paralelo hacer

$S_j \leftarrow \bigcup_{i=1}^n S_{j_i}$ {Recibir fragmentos S_j de los nodos S}
 $T_j \leftarrow R_j \text{ JP } S_j$ fin para {sonda S_j para cada tupla de R_j }

fin

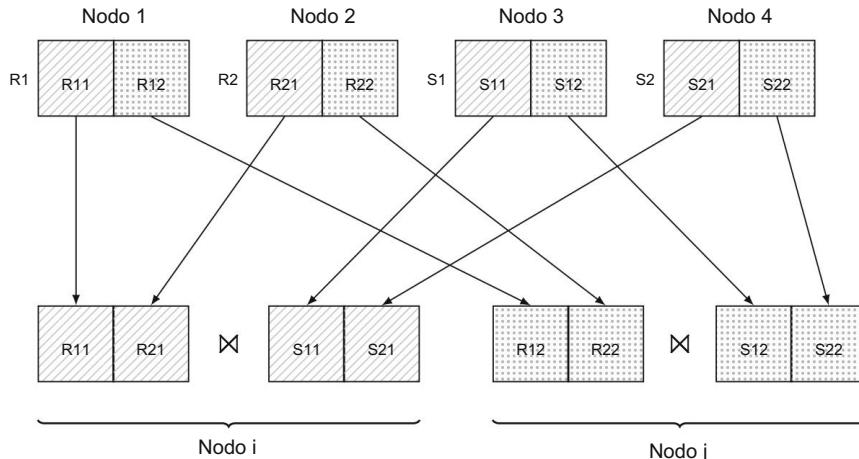


Fig. 8.12 Ejemplo de unión hash paralela

El algoritmo de unión hash paralela suele ser mucho más eficiente que el algoritmo de unión de bucles anidados paralelos, ya que requiere menos transferencia de datos y menos procesamiento local de unión en la fase de sondeo. Además, una relación, por ejemplo, R, puede estar ya particionada mediante hash en el atributo de unión. En este caso, no se requiere una fase de compilación y los fragmentos S simplemente se envían asociativamente a los nodos R correspondientes. También suele ser más eficiente que el algoritmo de unión de ordenación-fusión paralela. Sin embargo, este último algoritmo sigue siendo útil, ya que genera una relación de resultados ordenada según el atributo de unión.

El problema con el algoritmo de unión hash paralela y sus múltiples variantes es que la distribución de datos en el atributo de unión puede estar sesgada, lo que provoca un desequilibrio de carga. Analizamos soluciones a este problema en la Sección [8.5.2](#).

Variantes

Los algoritmos básicos de unión paralela se han utilizado en numerosas variantes, en particular para el procesamiento adaptativo de consultas o para aprovechar al máximo la memoria principal y los procesadores multinúcleo. A continuación, analizamos estas extensiones.

Al considerar el procesamiento adaptativo de consultas (véase la sección [4.6](#)), el reto reside en ordenar dinámicamente los operadores de unión segmentados en tiempo de ejecución, mientras fluyen tuplas de diferentes relaciones. Idealmente, cuando llega una tupla de una relación que participa en una unión, debería enviarse a un operador de unión para su procesamiento sobre la marcha. Sin embargo, la mayoría de los algoritmos de unión no pueden procesar algunas tuplas entrantes sobre la marcha debido a su asimetría con respecto al procesamiento de las tuplas internas y externas. Consideremos, por ejemplo, PHJ: la relación interna se lee completamente durante la fase de construcción para construir una tabla hash, mientras que las tuplas de la relación externa pueden segmentarse durante la fase de sondeo. Por lo tanto, una tupla interna entrante no puede procesarse sobre la marcha, ya que debe almacenarse en la tabla hash y el procesamiento solo será posible cuando se haya construido toda la tabla hash. De forma similar, el algoritmo de unión de bucle anidado es asimétrico, ya que solo debe leerse completamente la relación interna para cada tupla de la relación externa. Los algoritmos de unión con algún tipo de asimetría ofrecen pocas posibilidades de alternar las relaciones de entrada entre roles internos y externos. Por lo tanto, para flexibilizar el orden en que se consumen las entradas de unión, se requieren algoritmos de unión simétricos, que permiten cambiar el rol de las relaciones en una unión sin generar resultados incorrectos.

El ejemplo anterior de algoritmo de unión simétrica es la unión hash simétrica, que utiliza dos tablas hash, una para cada relación de entrada. Las fases tradicionales de construcción y sondeo del algoritmo básico de unión hash se intercalan. Cuando llega una tupla, se utiliza para sondear la tabla hash correspondiente a la otra relación y encontrar tuplas coincidentes. Luego, se inserta en su tabla hash correspondiente para que las tuplas de la otra relación que lleguen posteriormente puedan unirse. De este modo, cada tupla que llega puede procesarse sobre la marcha. Otro algoritmo de unión simétrica popular es la unión de ondulación, que es una generalización del algoritmo de unión de bucle anidado, donde los roles de la relación interna y externa se alternan continuamente durante la ejecución de la consulta. La idea principal es mantener el estado de sondeo de cada relación de entrada, con un puntero que indica la última tupla utilizada para sondear la otra relación. En cada punto de comutación, se produce un cambio de roles.

Se produce una conexión entre las relaciones internas y externas. En este punto, la nueva relación externa comienza a analizar la entrada interna desde su posición de puntero hasta un número específico de tuplas. La relación interna, a su vez, se escanea desde su primera tupla hasta su posición de puntero menos 1. El número de tuplas procesadas en cada etapa de la relación externa proporciona la tasa de alternancia y puede monitorizarse adaptativamente.

Aprovechar la memoria principal de los procesadores también es importante para el rendimiento de los algoritmos de unión paralela. El algoritmo de unión hash híbrido mejora la unión hash Grace al aprovechar la memoria disponible para albergar una partición completa (denominada partición 0) durante el particionado, evitando así los accesos al disco. Otra variación consiste en modificar la fase de compilación para que las tablas hash resultantes quepan en la memoria principal del procesador. Esto mejora significativamente el rendimiento, ya que se reduce el número de fallos de caché al sondear la tabla hash. La misma idea se utiliza en el algoritmo de unión hash radix para procesadores multinúcleo, donde el acceso a la memoria de un núcleo es mucho más rápido que a la memoria compartida remota. Se utiliza un esquema de particionamiento multipaso para dividir ambas relaciones de entrada en particiones disjuntas según el atributo de unión, de modo que quepan en las memorias de los núcleos. A continuación, se construyen tablas hash sobre cada partición de la relación interna y se sondan utilizando los datos de la partición correspondiente de la relación externa. La unión de ordenación por fusión paralela, generalmente considerada inferior a la unión hash paralela, también puede optimizarse para procesadores multinúcleo.

8.4.2 Optimización de consultas paralelas

La optimización de consultas paralelas presenta similitudes con el procesamiento de consultas distribuidas. Sin embargo, se centra mucho más en aprovechar tanto el paralelismo intraoperador (utilizando los algoritmos descritos anteriormente) como el interoperador. Como cualquier optimizador de consultas, un optimizador de consultas paralelo consta de tres componentes: un espacio de búsqueda, un modelo de costes y una estrategia de búsqueda. En esta sección, describimos las técnicas de paralelismo para estos componentes.

8.4.2.1 Espacio de búsqueda

Los planes de ejecución se resumen mediante árboles de operadores, que definen el orden de ejecución. Estos árboles se enriquecen con anotaciones que indican aspectos adicionales de la ejecución, como el algoritmo de cada operador.

En un SGBD paralelo, un aspecto importante de la ejecución que deben reflejar las anotaciones es que dos operadores subsiguientes pueden ejecutarse en secuencia. En este caso, el segundo operador puede comenzar antes de que el primero se complete. En otras palabras, el segundo operador empieza a consumir tuplas en cuanto el primero las produce .

Las ejecuciones pipeline no requieren que se materialicen relaciones temporales, es decir, no se almacena un nodo trié correspondiente a un operador ejecutado en pipeline.

Algunos operadores y algoritmos requieren el almacenamiento de un operando. Por ejemplo, en PHJ (Algoritmo 8.2), en la fase de construcción, se construye una tabla hash en

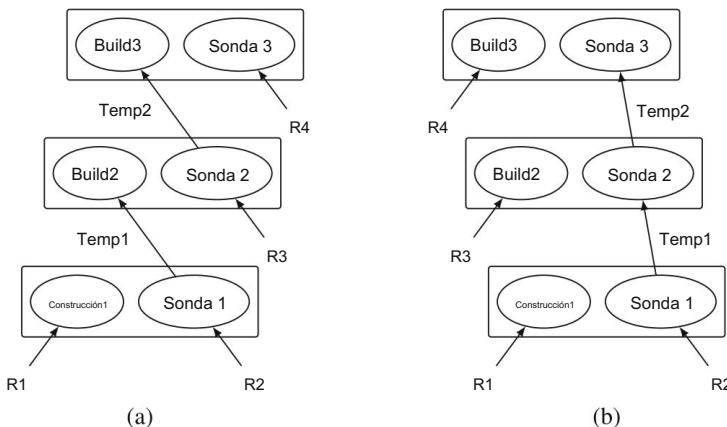


Fig. 8.13 Dos árboles de unión hash con diferente programación. (a) Sin canalización. (b) Canalización de R2, Temp1 y Temp2.

Paralelo en el atributo de unión de la relación más pequeña. En la fase de sondeo, se escanea secuencialmente la relación más grande y se consulta la tabla hash para cada una de sus tuplas.

Por lo tanto, las anotaciones de pipeline y almacenadas restringen la programación de los planes de ejecución al dividir un trie de operadores en subárboles no superpuestos, correspondientes a las fases de ejecución. Los operadores de pipeline se ejecutan en la misma fase, generalmente denominada cadena de pipeline, mientras que una indicación de almacenamiento establece el límite entre una fase y la siguiente.

Ejemplo 8.4. La Figura 8.13 muestra dos árboles de ejecución, uno sin pipeline (Fig. 8.13a) y otro con pipeline (Fig. 8.13b). En la Fig. 8.13a, la relación temporal Temp1 debe generarse completamente y la tabla hash en Build2 debe generarse antes de que Probe2 pueda empezar a consumir R3. Lo mismo ocurre con Temp2, Build3 y Probe3. Por lo tanto, el árbol se ejecuta en cuatro fases consecutivas: (1) generar la tabla hash de R1, (2) sondearla con R2 y generar la tabla hash de Temp1, (3) sondearla con R3 y generar la tabla hash de Temp2, (4) sondearla con R3 y generar el resultado. La Figura 8.13b muestra la ejecución de un pipeline. El trie se puede ejecutar en dos fases si hay suficiente memoria disponible para construir las tablas hash: (1) construir las tablas para R1, R3 y R4, (2) ejecutar Probe1, Probe2 y Probe3 en la canalización.

El conjunto de nodos donde se almacena una relación se denomina su origen. El origen de un operador es el conjunto de nodos donde se ejecuta y debe ser el origen de sus operandos para que el operador pueda acceder a su operando. Para operadores binarios como join, esto podría implicar la repartición de uno de los operandos. El optimizador podría incluso, en ocasiones, considerar la repartición de ambos operandos como de interés. Los árboles de operadores incluyen anotaciones de ejecución para indicar la repartición.

La Figura 8.14 muestra cuatro árboles de operadores que representan planes de ejecución para una unión de tres vías. Los árboles de operadores pueden ser lineales (es decir, al menos un operando de cada nodo de unión es una relación base) o ramificados. Es conveniente representar las relaciones segmentadas como...

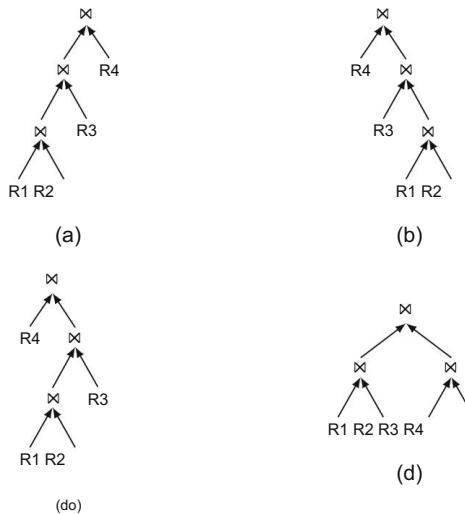


Fig. 8.14 Planes de ejecución como árboles de operadores. (a) Izquierda profunda. (b) Derecha profunda. (c) Zigzag. (d) Tupido.

Entrada del lado derecho de un operador. Por lo tanto, los árboles de profundidad derecha expresan una segmentación completa, mientras que los de profundidad izquierda expresan la materialización completa de todos los resultados intermedios. Por lo tanto, suponiendo suficiente memoria para almacenar las relaciones del lado izquierdo, los árboles de profundidad derecha largos son más eficientes que los árboles de profundidad izquierda correspondientes. En un trie de profundidad izquierda como el de la Fig. 8.14a, solo el último operador puede consumir su relación de entrada derecha en la segmentación, siempre que esta pueda almacenarse completamente en la memoria principal.

Los formatos de trie paralelos, distintos de los de profundidad izquierda o derecha, también son interesantes. Por ejemplo, los árboles frondosos (Fig. 8.14d) son los únicos que permiten paralelismo independiente y cierto paralelismo de canalización. El paralelismo independiente es útil cuando las relaciones se dividen en hogares disjuntos. Supongamos que las relaciones de la Fig. 8.14d se dividen de tal manera que R1 y R2 tienen el mismo hogar h1 , y R3 y R4 tienen el mismo hogar h2 , que es diferente de h1 . Entonces, las dos uniones de las relaciones base podrían ser ejecutadas independientemente en paralelo por el conjunto de nodos que constituye h1 y h2.

Cuando el paralelismo de pipelines resulta beneficioso, los árboles en zigzag, formatos intermedios entre los árboles de profundidad izquierda y derecha, a veces superan a los de profundidad derecha gracias a un mejor uso de la memoria principal. Una heurística razonable es favorecer los árboles de profundidad derecha o en zigzag cuando las relaciones están parcialmente fragmentadas en casas disjuntas y las relaciones intermedias son bastante grandes. En este caso, los árboles frondosos suelen necesitar más fases y su ejecución es más larga. Por el contrario, cuando las relaciones intermedias son pequeñas, la segmentación no es muy eficiente debido a la dificultad de equilibrar la carga entre las etapas de la segmentación.

Con los árboles de operadores anteriores, los operadores deben capturar el paralelismo, lo que requiere repartir las relaciones de entrada. Esto se exemplifica en el algoritmo PHJ (véase la sección 8.4.1.2), donde las relaciones de entrada se partitionan según la misma función hash aplicada al atributo de unión, seguida de una unión paralela en las particiones locales.

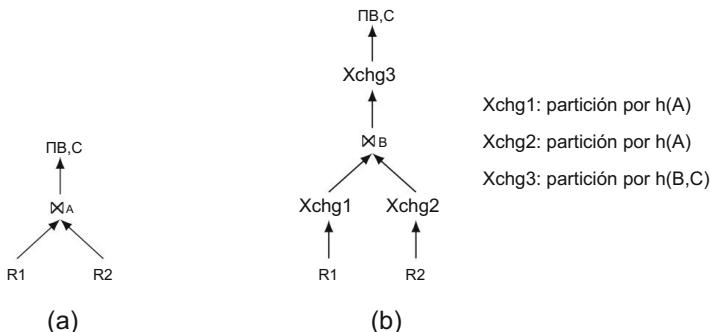


Fig. 8.15 Operador trie con operadores de intercambio. (a) Operador secuencial trie. (b) Operador paralelo trie.

Para facilitar la navegación del optimizador en el espacio de búsqueda, la repartición de datos se puede encapsular en un operador de intercambio. Dependiendo de cómo se realice la partición, podemos tener diferentes operadores de intercambio, como la partición hash, la partición por rango o la replicación de datos a varios nodos. Algunos ejemplos de usos de los operadores de intercambio son:

- Unión hash paralela: partición hash de las relaciones de entrada en el atributo de unión seguida de una unión local; • Unión de bucle anidado paralelo: replicación de la relación interna en los nodos donde se partitiona la relación externa, seguida de una unión local; • Ordenamiento de rango paralelo: partición de rango seguida de un ordenamiento local.

La Figura 8.15 muestra un ejemplo del operador `trie` con operadores de intercambio. La operación de unión se realiza mediante la partición hash de las relaciones de entrada en A (operadores `Xchg1` y `Xchg2`), seguida de una unión local. Las operaciones de proyecto se realizan mediante la eliminación de duplicados mediante hash (operador `Xchg3`), seguida de una proyección local.

8.4.2.2 Modelo de costos

Recuerde que el modelo de costos del optimizador se encarga de estimar el costo de un plan de ejecución determinado. Consta de dos partes: una dependiente de la arquitectura y otra independiente de la arquitectura. La parte independiente de la arquitectura está constituida por las funciones de costo de los algoritmos de operador, por ejemplo, el bucle anidado para la unión y el acceso secuencial para la selección. Si ignoramos los problemas de concurrencia, solo difieren las funciones de coste para la repartición de datos y el consumo de memoria, que constituyen la parte dependiente de la arquitectura. De hecho, repartir las tuplas de una relación en un sistema sin recursos compartidos implica transferencias de datos a través de la interconexión, mientras que en sistemas con memoria compartida se reduce al hash. El consumo de memoria en el caso de memoria compartida se complica por el paralelismo entre operadores. En los sistemas de memoria compartida, todos los operadores leen y escriben datos a través de una memoria global, y es fácil comprobar si hay suficiente espacio para ejecutar.

Se ejecutan en paralelo, es decir, la suma del consumo de memoria de cada operador es menor que la memoria disponible. En el modelo sin memoria compartida, cada procesador tiene su propia memoria, y es importante saber qué operadores se ejecutan en paralelo en el mismo procesador. Por lo tanto, para simplificar, podemos asumir que el conjunto de procesadores (home) asignado a los operadores no se superpone; es decir, la intersección del conjunto de procesadores está vacía o los conjuntos son idénticos.

El tiempo total de un plan se puede calcular mediante una fórmula que simplemente suma todos los componentes de costo de CPU, E/S y comunicación, como en la optimización de consultas distribuidas. El tiempo de respuesta es más complejo, ya que debe considerar la canalización.

El tiempo de respuesta del plan p, programado en fases (cada una denotada por ph), es Se calcula de la siguiente manera:

$$RT(p) = \sum_{Op} (\max(Op) \cdot ph(\text{respT_ime}(Op) + \text{pipe_delay}(Op)) \cdot ph \cdot p)$$

$$+ \text{retraso_de_almacenamiento}(ph))$$

donde Op denota un operador, respT ime(Op) es el tiempo de respuesta de Op, pipe_delay(Op) es el período de espera de Op necesario para que el productor entregue las primeras tuplas de resultados (es igual a 0 si se almacenan las relaciones de entrada de Op), store_delay(ph) es el tiempo necesario para almacenar el resultado de salida de la fase ph (es igual a 0 si ph es la última fase, asumiendo que los resultados se entregan tan pronto como se producen).

Para estimar el costo de un plan de ejecución, el modelo de costos utiliza estadísticas de base de datos e información de la organización, como cardinalidades de relación y particiones, como en la optimización de consultas distribuidas.

8.4.2.3 Estrategia de búsqueda

La estrategia de búsqueda no necesita ser diferente de la optimización de consultas centralizada o distribuida. Sin embargo, el espacio de búsqueda tiende a ser mucho mayor debido a que existen más parámetros que impactan los planes de ejecución en paralelo, en particular, las anotaciones de pipeline y de almacenamiento. Por lo tanto, las estrategias de búsqueda aleatorias como la Mejora Iterativa y el Recocido Simulado generalmente superan a las estrategias de búsqueda deterministas tradicionales en la optimización de consultas en paralelo. Otro enfoque interesante, aunque simple, para reducir el espacio de búsqueda es la estrategia de optimización en dos fases propuesta para XPRS, un DBMS paralelo de memoria compartida. Primero, en tiempo de compilación, se genera el plan de consulta óptimo basado en un modelo de costos centralizado. Luego, en tiempo de ejecución, se consideran parámetros de tiempo de ejecución como el tamaño de búfer disponible y el número de procesadores libres para parallelizar el plan de consulta. Se ha demostrado que este enfoque casi siempre produce planes óptimos.

8.5 Equilibrio de carga

Un buen balanceo de carga es crucial para el rendimiento de un sistema paralelo. El tiempo de respuesta de un conjunto de operadores paralelos es el del más largo. Por lo tanto, minimizar el tiempo del más largo es importante para minimizar el tiempo de respuesta.

Equilibrar la carga de los diferentes nodos también es esencial para maximizar el rendimiento.

Aunque el optimizador de consultas paralelas incorpora decisiones sobre cómo ejecutar un plan de ejecución paralela, el equilibrio de carga puede verse afectado por diversos problemas que surgen durante la ejecución. Se pueden obtener soluciones a estos problemas a nivel intraoperador e interoperador. En esta sección, analizamos estos problemas de ejecución paralela y sus soluciones.

8.5.1 Problemas de ejecución paralela

Los principales problemas introducidos por la ejecución de consultas paralelas son la inicialización, la interferencia y el sesgo.

Inicialización

Antes de la ejecución, es necesario un paso de inicialización. Este paso suele ser secuencial e incluye la creación e inicialización de tareas (o subprocesos), la inicialización de la comunicación, etc. Su duración es proporcional al grado de paralelismo y puede, de hecho, dominar el tiempo de ejecución de consultas simples, por ejemplo, una consulta de selección sobre una sola relación. Por lo tanto, el grado de paralelismo debe ajustarse según la complejidad de la consulta.

Se puede desarrollar una fórmula para estimar la aceleración máxima alcanzable durante la ejecución de un operador y deducir el número óptimo de procesadores. Consideremos la ejecución de un operador que procesa N tuplas con n procesadores.

Sea c el tiempo promedio de procesamiento de cada tupla y a la tiempo de inicialización por procesador. En el caso ideal, el tiempo de respuesta de la ejecución del operador es

$$\text{Tiempo de respuesta} = (a \cdot n) + \frac{c \cdot N}{n}$$

Por derivación, podemos obtener el número óptimo de procesadores que no se pueden asignar. y la aceleración máxima alcanzable (Speedmax).

$$nopt = \frac{c \cdot N}{a} \quad \text{Velocidad máxima} = \frac{nopt}{2}$$

El número óptimo de procesadores ($nopt$) es independiente de n y solo depende del tiempo total de procesamiento y del tiempo de inicialización. Por lo tanto, se maximiza el grado de

El paralelismo para un operador, por ejemplo, utilizando todos los procesadores disponibles, puede perjudicar la aceleración debido a la sobrecarga de inicialización.

Interferencia

Una ejecución altamente paralela puede verse ralentizada por interferencias. Estas se producen cuando varios procesadores acceden simultáneamente al mismo recurso, hardware o software. Un ejemplo típico de interferencia de hardware es la contención creada en la interconexión de un sistema de memoria compartida. Al aumentar el número de procesadores, aumenta el número de conflictos en la interconexión, lo que limita la extensibilidad de los sistemas de memoria compartida. Una solución a estas interferencias es duplicar los recursos compartidos. Por ejemplo, la interferencia en el acceso a discos puede eliminarse añadiendo varios discos y particionando las relaciones.

La interferencia de software se produce cuando varios procesadores desean acceder a datos compartidos. Para evitar incoherencias, se utilizan variables de exclusión mutua para proteger los datos compartidos, bloqueando así el acceso a todos los procesadores, excepto uno. Esto es similar a los algoritmos de control de concurrencia basados en bloqueos (véase el capítulo 5). Sin embargo, las variables compartidas pueden convertirse en un cuello de botella en la ejecución de consultas, creando puntos calientes. Un ejemplo típico de interferencia de software es el acceso a estructuras internas de bases de datos, como índices y búferes. Para simplificar, las versiones anteriores de los sistemas de bases de datos estaban protegidas por una única variable de exclusión mutua, lo que generaba una gran sobrecarga.

Una solución general a la interferencia de software es partitionar el recurso compartido en varios recursos independientes, cada uno protegido por una variable de exclusión mutua diferente. De esta manera, se puede acceder a dos recursos independientes en paralelo, lo que reduce la probabilidad de interferencia. Para reducir aún más la interferencia en un recurso independiente (por ejemplo, una estructura de índice), se puede utilizar la replicación. Por lo tanto, el acceso a los recursos replicados también se puede parallelizar.

Sesgar

Los problemas de equilibrio de carga pueden surgir con el paralelismo entre operadores (variación en el tamaño de la partición), es decir, la desviación de datos, y el paralelismo entre operadores (variación en la complejidad de los operadores).

Los efectos de la distribución sesgada de datos en una ejecución paralela se pueden clasificar de la siguiente manera: la desviación del valor del atributo (AVS) es una desviación inherente a los datos (por ejemplo, hay más ciudadanos en París que en Waterloo), mientras que la desviación de la ubicación de tuplas (TPS) es la desviación introducida cuando los datos se partitionan inicialmente (por ejemplo, con partición de rango). El sesgo de selectividad (SS) se introduce cuando existe variación en la selectividad de los predicados de selección en cada nodo. El sesgo de redistribución (RS) ocurre en el paso de redistribución entre dos operadores. Es similar al TPS. Finalmente, el sesgo de producto de unión (JPS) ocurre porque la selectividad de unión puede variar entre nodos. La Figura 8.16 ilustra esta clasificación en una consulta sobre dos relaciones R y S con particiones deficientes.

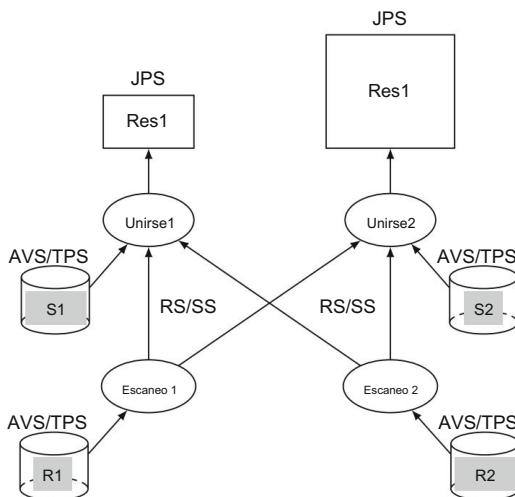


Fig. 8.16 Ejemplo de sesgo de datos

Las casillas son proporcionales al tamaño de las particiones correspondientes. Este particionamiento deficiente se debe a los datos (AVS) o a la función de particionamiento (TPS). Por lo tanto, los tiempos de procesamiento de las dos instancias Scan1 y Scan2 no son iguales. El caso del operador de unión es aún peor. En primer lugar, el número de tuplas recibidas varía entre instancias debido a una redistribución deficiente de las particiones de R (RS) o a una selectividad variable según la partición de R procesada (SS). Finalmente, el tamaño desigual de las particiones S (AVS/TPS) genera diferentes tiempos de procesamiento para las tuplas enviadas por el operador de escaneo, y el tamaño del resultado varía entre particiones debido a la selectividad de unión (JPS).

8.5.2 Equilibrio de carga intraoperador

Un buen balanceo de carga intraoperador depende del grado de paralelismo y la asignación de procesadores para el operador. Para algunos algoritmos, p. ej., PHJ, estos parámetros no están restringidos por la ubicación de los datos. Por lo tanto, la ubicación del operador (el conjunto de procesadores donde se ejecuta) debe decidirse cuidadosamente. El problema de sesgo dificulta que el optimizador de consultas paralelas tome esta decisión estáticamente (en tiempo de compilación), ya que requeriría un modelo de costos muy preciso y detallado. Por lo tanto, las principales soluciones se basan en técnicas adaptativas o especializadas que pueden incorporarse en un optimizador de consultas híbrido. A continuación, describimos estas técnicas en el contexto del procesamiento de uniones paralelas, que ha recibido mucha atención. Para simplificar, asumimos que a cada operador se le asigna una ubicación según lo decide el procesador de consultas (ya sea estáticamente o justo antes de la ejecución).

Técnicas adaptativas

La idea principal es decidir estáticamente una asignación inicial de procesadores al operador (mediante un modelo de costes) y, en tiempo de ejecución, adaptar la asignación de carga mediante la reasignación de carga. Un enfoque sencillo para la reasignación de carga consiste en detectar las particiones sobredimensionadas y volver a particionarlas en varios procesadores (entre los ya asignados a la operación) para aumentar el paralelismo. Este enfoque se ha generalizado para permitir un ajuste más dinámico del grado de paralelismo.

Utiliza operadores de control específicos en el plan de ejecución para detectar si las estimaciones estáticas para tamaños de resultados intermedios diferirán de los valores en tiempo de ejecución. Durante la ejecución, si la diferencia entre la estimación y el valor real es suficientemente alta, el operador de control realiza una redistribución de la relación para evitar sesgos en el producto de unión y en la redistribución. Las técnicas adaptativas son útiles para mejorar el balanceo de carga intraoperador en todo tipo de arquitecturas paralelas. Sin embargo, la mayor parte del trabajo se ha realizado en el contexto de no compartido, donde los efectos del desequilibrio de carga son más graves en el rendimiento. DBS3 ha sido pionero en el uso de una técnica adaptativa basada en la partición de la relación (como en no compartido) para memoria compartida. Al reducir la interferencia del procesador, esta técnica proporciona un excelente balanceo de carga para el paralelismo intraoperador.

Técnicas especializadas

Los algoritmos de unión paralela pueden especializarse para gestionar la asimetría. Un enfoque consiste en utilizar múltiples algoritmos de unión, cada uno especializado para un grado diferente de asimetría, y determinar, en tiempo de ejecución, cuál es el mejor. Esto se basa en dos técnicas principales: partición por rangos y muestreo. La partición por rangos se utiliza en lugar de la partición hash (en el algoritmo de unión hash paralela) para evitar la asimetría por redistribución de la relación de construcción. De este modo, los procesadores pueden obtener particiones de igual número de tuplas, correspondientes a diferentes rangos de valores de los atributos de unión. Para determinar los valores que delimitan los valores de rango, se utiliza el muestreo de la relación de construcción para generar un histograma de los valores de los atributos de unión, es decir, el número de tuplas para cada valor de atributo. El muestreo también es útil para determinar qué algoritmo y qué relación utilizar para la construcción o el sondeo. Mediante estas técnicas, el algoritmo de unión hash paralela puede adaptarse para gestionar la asimetría de la siguiente manera:

1. Muestrear la relación de construcción para determinar los rangos de partición.
2. Redistributions la relación de construcción entre los procesadores mediante los rangos. Cada procesador crea una tabla hash que contiene las tuplas entrantes.
3. Redistributions la relación de sondeo entre los procesadores utilizando los mismos rangos. Por cada tupla recibida, cada procesador sondea la tabla hash para realizar la unión.

Este algoritmo puede mejorarse aún más para gestionar un sesgo elevado mediante técnicas adicionales y diferentes estrategias de asignación de procesadores. Un enfoque similar consiste en modificar los algoritmos de unión insertando un paso de programación encargado de redistribuir la carga en tiempo de ejecución.

8.5.3 Equilibrio de carga entre operadores

Para lograr un buen balanceo de carga a nivel interoperador, es necesario elegir, para cada operador, cuántos y cuáles procesadores asignar para su ejecución. Esto debe hacerse considerando el paralelismo de pipeline, que requiere comunicación interoperador. Esto es más difícil de lograr en entornos sin recursos compartidos por las siguientes razones: primero, el grado de paralelismo y la asignación de procesadores a los operadores, cuando se deciden en la fase de optimización paralela, se basan en un modelo de costos posiblemente impreciso. segundo, la elección del grado de paralelismo está sujeta a errores, ya que tanto los procesadores como los operadores son entidades discretas.

Finalmente, los procesadores asociados con los últimos operadores de una cadena de pipeline pueden permanecer inactivos durante un tiempo considerable. Esto se conoce como el problema de retardo de pipeline.

El enfoque principal en el modelo sin recursos compartidos consiste en determinar dinámicamente (justo antes de la ejecución) el grado de paralelismo y la localización de los procesadores para cada operador. Por ejemplo, el algoritmo de coincidencia de velocidad utiliza un modelo de costes para igualar la velocidad de producción y consumo de las tuplas. Este algoritmo sirve de base para seleccionar el conjunto de procesadores que se utilizará para la ejecución de consultas (en función de la memoria disponible, la CPU y el uso del disco). Existen muchos otros algoritmos para la selección del número y la localización de los procesadores, por ejemplo, maximizando el uso de varios recursos mediante estadísticas sobre su uso.

En discos y memorias compartidas, existe mayor flexibilidad, ya que todos los procesadores tienen el mismo acceso a los discos. Al no requerirse particionamiento físico, cualquier procesador puede asignarse a cualquier operador. En particular, a un procesador se le pueden asignar todos los operadores de la misma cadena de pipelines, sin paralelismo entre operadores. Sin embargo, este paralelismo es útil para ejecutar cadenas de pipelines independientes. El enfoque propuesto en XPRS para memoria compartida permite la ejecución paralela de cadenas de pipelines independientes, denominadas tareas. La idea principal es combinar tareas de E/S y de CPU para aumentar la utilización de los recursos del sistema.

Antes de su ejecución, una tarea se clasifica como dependiente de E/S o dependiente de la CPU utilizando la información del modelo de costes, como se indica a continuación. Supongamos que, si se ejecuta secuencialmente, la tarea t genera accesos al disco a una tasa $I\text{Rate}(t)$, es decir, en número de accesos al disco por segundo. Consideremos un sistema de memoria compartida con n procesadores y un ancho de banda total de disco de B (número de accesos al disco por segundo). La tarea t se define como dependiente de E/S si $I\text{Rate}(t) > B/n$ y, en caso contrario, dependiente de la CPU. Las conversaciones dependientes de la CPU y de E/S pueden ejecutarse en paralelo en su punto óptimo de equilibrio E/S-CPU. Esto se logra ajustando dinámicamente el grado de paralelismo intraoperador de las tareas para alcanzar la máxima utilización de recursos.

8.5.4 Equilibrio de carga entre consultas

El balanceo de carga intraconsulta debe combinar el paralelismo intraoperador e interoperador. Hasta cierto punto, dada una arquitectura paralela, las técnicas de balanceo de carga intraoperador o interoperador que acabamos de presentar pueden combinarse. Sin embargo, en clústeres sin recursos compartidos con nodos de memoria compartida (o procesadores multinúcleo),

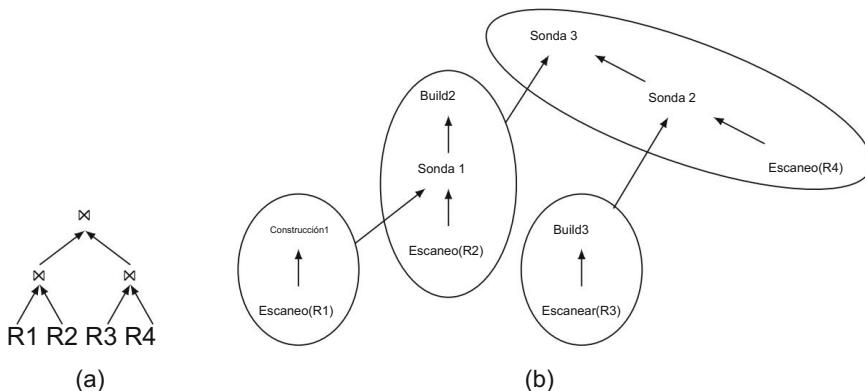


Fig. 8.17 Un trie de unión y un trie de operador asociado. (a) Trie de unión. (b) Trie de operador (las elipses son cadenas de tuberías)

Los problemas de balanceo de carga se agravan porque deben abordarse a dos niveles: localmente entre los procesadores o núcleos de cada nodo de memoria compartida (nodo SM) y globalmente entre todos los nodos. Ninguno de los enfoques de balanceo de carga intraoperador e interoperador que se acaban de describir puede extenderse fácilmente para abordar este problema. Las estrategias de balanceo de carga para sistemas sin memoria compartida experimentarían un agravamiento de los problemas (p. ej., complejidad e inexactitud del modelo de costes). Por otro lado, la adaptación de soluciones dinámicas desarrolladas para sistemas de memoria compartida generaría una alta sobrecarga de comunicación.

Una solución general para el balanceo de carga es el modelo de ejecución llamado Procesamiento Dinámico (PD). La idea fundamental es que la consulta se descompone en unidades autónomas de procesamiento secuencial, cada una de las cuales puede ser realizada por cualquier procesador. Intuitivamente, un procesador puede migrar horizontalmente (paralelismo intraoperador) y verticalmente (paralelismo interoperador) a lo largo de los operadores de consulta. Esto minimiza la sobrecarga de comunicación del balanceo de carga entre nodos al maximizar el balanceo de carga intraoperador e interoperador dentro de los nodos de memoria compartida. La entrada al modelo de ejecución es un plan de ejecución paralela producido por el optimizador, es decir, un trie de operadores con programación de operadores y asignación de recursos computacionales a los operadores. Las restricciones de programación de operadores expresan un orden parcial entre los operadores de la consulta: Op1 → Op2 indica que el operador Op1 no puede comenzar antes del operador Op2.

Ejemplo 8.5 La Figura 8.17 muestra un trie de unión con cuatro relaciones R1, R2, R3 y R4, y el trie de operador correspondiente con las cadenas de canalización claramente identificadas.

Suponiendo que se utiliza una unión hash paralela, las restricciones de programación del operador están entre los operadores de compilación y sondeo asociados:

Build1	Probe1
Build2	Probe3
Build3	Probe2

También existen heurísticas de programación entre operadores de diferentes tuberías. cadenas que se derivan de las restricciones de programación:

Heurística 1: Compilación 1 Escanear(R2), Compilación 3 Escanear(R4), Compilación 2 Escanear(R3)
 Heurística 2: Construir 2 Escanear (R3)

Suponiendo tres nodos SM i, j y k con R1 almacenado en el nodo i, R2 y R3 en el nodo y R4 en el nodo k, podemos tener los siguientes operadores homes: j,

inicio (Escanear(R1)) = i
 inicio (Compilación1, Sonda1, Escanear(R2), Escanear(R3)) = j
 inicio (Escanear(R4)) = k
 inicio (Compilación2, Compilación3, Sonda2, Sonda3) = j y k

Dado un operador trie de este tipo, el problema consiste en generar una ejecución que minimice el tiempo de respuesta. Esto se puede lograr mediante un mecanismo de balanceo de carga dinámico en dos niveles: (i) dentro de un nodo SM, el balanceo de carga se logra mediante una comunicación rápida entre procesos; (ii) entre nodos SM, se requiere una comunicación de paso de mensajes más costosa. Por lo tanto, el problema consiste en crear un modelo de ejecución que maximice el uso del balanceo de carga local y minimice el uso del balanceo de carga global (mediante el paso de mensajes).

Llamamos activación a la unidad más pequeña de procesamiento secuencial que no puede ser particionada. La propiedad principal del modelo DP es permitir que cualquier procesador procese cualquier activación de su nodo SM. Por lo tanto, no existe una asociación estática entre hilos y operadores. Esto proporciona un buen equilibrio de carga para el paralelismo intraoperador e interoperador dentro de un nodo SM, minimizando así la necesidad de equilibrio de carga global, es decir, cuando ya no hay trabajo que realizar en un nodo SM.

El modelo de ejecución de DP se basa en algunos conceptos: activaciones, activación colas y subprocessos.

Activaciones

Una activación representa una unidad secuencial de trabajo. Dado que cualquier activación puede ser ejecutada por cualquier hilo (por cualquier procesador), las activaciones deben ser autocontenido y hacer referencia a toda la información necesaria para su ejecución: el código a ejecutar y los datos a procesar. Se pueden distinguir dos tipos de activaciones: activaciones de disparador y activaciones de datos. Una activación de disparador se utiliza para iniciar la ejecución de un operador de hoja, es decir, escaneo. Se representa mediante un par (Operador, Partición) que hace referencia al operador de escaneo y la partición de la relación base a escanear. Una activación de datos describe una tupla producida en modo pipeline. Se representa mediante un triple (Operador, Tupla, Partición) que hace referencia al operador a procesar. Para un operador de compilación, la activación de datos especifica que la tupla debe insertarse en la tabla hash del contenedor y para un operador de sondeo, que la tupla debe sondarse con

La tabla hash de la partición. Aunque las activaciones son autónomas, solo pueden ejecutarse en el nodo SM donde se encuentran los datos asociados (tablas hash o relaciones base).

Colas de activación

El movimiento de activaciones de datos a lo largo de las cadenas de pipeline se realiza mediante colas de activación asociadas a operadores. Si el productor y el consumidor de una activación se encuentran en el mismo nodo SM, el movimiento se realiza mediante memoria compartida. De lo contrario, requiere el paso de mensajes. Para unificar el modelo de ejecución, se utilizan colas para activaciones de disparadores (entradas para operadores de escaneo), así como para activaciones de tuplas (entradas para operadores de compilación o sondeo). Todos los subprocessos tienen acceso sin restricciones a todas las colas ubicadas en su nodo SM. Gestionar un número pequeño de colas (p. ej., una para cada operador) puede generar interferencias. Para reducirlas, se asocia una cola a cada subprocesso que trabaja en un operador. Tenga en cuenta que un mayor número de colas probablemente intercambiaría interferencias por sobrecarga en la gestión de colas. Para reducir aún más la interferencia sin aumentar el número de colas, cada subprocesso tiene acceso prioritario a un conjunto distinto de colas, denominadas colas principales. Por lo tanto, un subprocesso siempre intenta consumir primero las activaciones en sus colas principales. Durante la ejecución, las restricciones de programación de operadores pueden implicar que un operador se bloquee hasta el final de otros operadores (los operadores bloqueantes). Por lo tanto, una cola para un operador bloqueado también está bloqueada; es decir, sus activaciones no se pueden consumir, pero sí se pueden producir si el operador productor no está bloqueado. Cuando todos sus operadores bloqueantes terminan, la cola bloqueada se vuelve consumible; es decir, los subprocessos pueden consumir sus activaciones. Esto se ilustra en la Fig. 8.18 con una instantánea de ejecución para el operador *trie* de la Fig. 8.17.

Trapos

Una estrategia sencilla para lograr un buen equilibrio de carga dentro de un nodo SM consiste en asignar un número de subprocessos mucho mayor que el de procesadores y dejar que el sistema operativo se encargue de la programación de subprocessos. Sin embargo, esta estrategia genera un alto número de llamadas al sistema debido a la programación de subprocessos y a las interferencias. En lugar de depender del sistema operativo para el equilibrio de carga, es posible asignar solo un subprocesso por procesador y por consulta. Esto es posible gracias a que cualquier subprocesso puede ejecutar cualquier operador asignado a su nodo SM. La ventaja de esta estrategia de asignación de un subprocesso por procesador es que reduce significativamente la sobrecarga por interferencias y sincronización, siempre que un subprocesso nunca se bloquee.

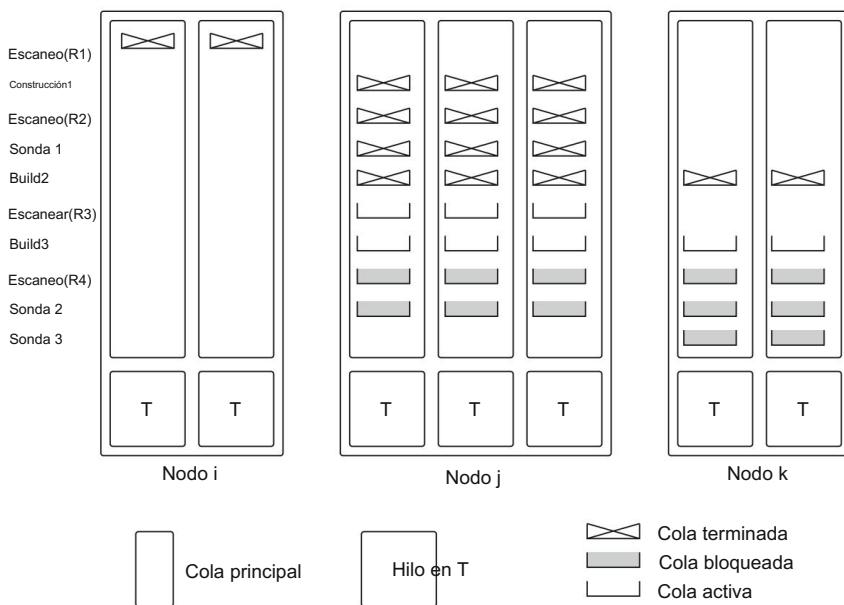


Fig. 8.18 Instantánea de una ejecución

El equilibrio de carga dentro de un nodo SM se logra asignando todas las colas de activación en un segmento de memoria compartida y permitiendo que todos los subprocesos consuman activaciones en cualquier cola. Para limitar la interferencia de los subprocesos, cada subproceso consumirá la mayor cantidad posible de su conjunto de colas principales antes de considerar las demás colas del nodo SM. Por lo tanto, un subproceso queda inactivo solo cuando ya no hay activación de ningún operador, lo que significa que no hay más trabajo que realizar en su nodo SM que está inactivo.

Cuando un nodo SM se queda sin recursos, compartimos la carga de otro nodo SM adquiriendo parte de su carga de trabajo. Sin embargo, la adquisición de activaciones (mediante el paso de mensajes) genera sobrecarga de comunicación. Además, la adquisición de activaciones no es suficiente, ya que también deben adquirirse los datos asociados, es decir, las tablas hash. Por lo tanto, el beneficio de adquirir activaciones y datos debe estimarse dinámicamente.

El nivel de balanceo de carga depende del número de operadores que se ejecutan simultáneamente, lo que permite encontrar trabajo compartido en caso de tiempos de inactividad. Se puede aumentar el número de operadores simultáneos permitiendo la ejecución simultánea de varias cadenas de pipeline o utilizando algoritmos de unión hash no bloqueantes, que permiten la ejecución simultánea de todos los operadores del trie con gran densidad de bits. Por otro lado, ejecutar más operadores simultáneamente puede aumentar el consumo de memoria. La programación estática de operadores, como la que proporciona el optimizador, debería evitar el desbordamiento de memoria y solucionar este problema.

8.6 Tolerancia a fallos

En esta sección, analizamos qué sucede cuando se producen fallos. Estos fallos plantean varios problemas. El primero es cómo mantener la consistencia a pesar de ellos.

En segundo lugar, en el caso de las transacciones pendientes, está el problema de cómo realizar la comutación por error.

En tercer lugar, cuando se reintroduce una réplica fallida (después de la recuperación) o se introduce una réplica nueva en el sistema, es necesario recuperar el estado actual de la base de datos.

La principal preocupación es cómo lidiar con los fallos. Para empezar, es necesario detectarlos. En los enfoques basados en la comunicación grupal (véase el capítulo 6), la detección de fallos la proporciona la comunicación grupal subyacente (normalmente basada en algún tipo de mecanismo de latido). Los cambios de membresía se notifican como eventos.¹ Al comparar la nueva membresía con la anterior, es posible saber qué réplicas han fallado. La comunicación grupal también garantiza que todas las réplicas conectadas comparten el mismo concepto de membresía. Para los enfoques que no se basan en la comunicación grupal, la detección de fallos puede delegarse a la capa de comunicación subyacente (p. ej., TCP/IP) o implementarse como un componente adicional de la lógica de replicación. Sin embargo, se necesita algún protocolo de acuerdo para garantizar que todas las réplicas conectadas comparten el mismo concepto de membresía de qué réplicas están operativas y cuáles no. De lo contrario, pueden surgir inconsistencias.

La API del cliente también debe detectar los fallos en el lado del cliente. Los clientes suelen conectarse a través de TCP/IP y pueden sospechar de nodos fallidos debido a conexiones interrumpidas. Ante un fallo en una réplica, la API del cliente debe descubrir una nueva réplica, restablecer la conexión y, en el caso más simple, retransmitir la última transacción pendiente a la réplica recién conectada. Dado que se requieren retransmisiones, podrían entregarse transacciones duplicadas. Esto requiere un mecanismo de detección y eliminación de transacciones duplicadas. En la mayoría de los casos, basta con tener un identificador único de cliente y un identificador único de transacción por cliente. Este último se incrementa con cada nueva transacción enviada. De esta forma, el clúster puede rastrear si una transacción de cliente ya se ha procesado y, de ser así, descartarla.

Una vez detectado un fallo en una réplica, se deben tomar varias medidas. Estas acciones forman parte del proceso de comutación por error, que debe redirigir las transacciones de un nodo fallido a otro nodo de réplica, de la forma más transparente posible para los clientes. La comutación por error depende en gran medida de si la réplica fallida era un nodo maestro.

Si falla una réplica no maestra, no es necesario realizar ninguna acción en el clúster. Los clientes con transacciones pendientes se conectan a un nuevo nodo de réplica y reenvían las últimas transacciones. Sin embargo, la pregunta interesante es qué definición de consistencia se proporciona. Recordemos que, en la sección 6.1, en una base de datos replicada, la serialización de una sola copia puede verse comprometida al serializar transacciones en diferentes nodos en orden inverso. Debido a la comutación por error, las transacciones también pueden procesarse de tal manera que se comprometa la serialización de una sola copia.

¹La literatura sobre comunicación grupal utiliza el término «cambio de vista» para referirse al cambio de membresía. En este artículo, no lo utilizaremos para evitar confusiones con el concepto de vista de base de datos .

En la mayoría de los enfoques de replicación, la comutación por error se gestiona cancelando todas las transacciones en curso para evitar estas situaciones. Sin embargo, esta forma de gestionar los fallos afecta a los clientes, que deben reenviar las transacciones canceladas. Dado que los clientes no suelen tener la capacidad transaccional de deshacer los resultados de una interacción conversacional, esto puede resultar muy complejo. El concepto de transacciones de alta disponibilidad hace que los fallos sean totalmente transparentes para los clientes, de modo que no observen cancelaciones de transacciones debido a fallos.

Las acciones a tomar en caso de fallo de una réplica maestra son más complejas, ya que se debe designar una nueva réplica maestra para que sustituya a la fallida. El nombramiento de una nueva réplica maestra debe ser acordado por todas las réplicas del clúster. En la replicación basada en grupos, gracias a la notificación de cambio de membresía, basta con aplicar una función determinista sobre la nueva membresía para asignar las réplicas maestras (todos los nodos reciben exactamente la misma lista de nodos activos y conectados).

Otro aspecto esencial de la tolerancia a fallos es la recuperación tras un fallo. La alta disponibilidad exige tolerar fallos y seguir proporcionando acceso consistente a los datos a pesar de ellos. Sin embargo, los fallos reducen el grado de redundancia del sistema, lo que degrada la disponibilidad y el rendimiento. Por lo tanto, es necesario reintroducir réplicas fallidas o nuevas en el sistema para mantener o mejorar la disponibilidad y el rendimiento. La principal dificultad radica en que las réplicas tienen estado y una réplica fallida puede haber perdido actualizaciones mientras estaba inactiva. Por lo tanto, una réplica fallida en recuperación necesita recibir las actualizaciones perdidas antes de poder empezar a procesar nuevas transacciones. Una solución es detener el procesamiento de transacciones. De este modo, se alcanza directamente un estado de reposo que puede ser transferido por cualquiera de las réplicas en funcionamiento a la que se está recuperando. Una vez que la réplica en recuperación ha recibido todas las actualizaciones perdidas, el procesamiento de transacciones puede reanudarse y todas las réplicas pueden procesar nuevas transacciones.

8.7 Clústeres de bases de datos

Un sistema de bases de datos paralelas suele implementar las funciones de gestión de datos paralelas de forma estrechamente acoplada, con todos los nodos homogéneos bajo el control total del SGBD paralelo. Una solución más sencilla (aunque no tan eficiente) consiste en utilizar un clúster de bases de datos, que consiste en un conjunto de bases de datos autónomas, cada una gestionada por un SGBD estándar. Una diferencia importante con un SGBD paralelo implementado en un clúster es el uso de un SGBD de "caja negra" en cada nodo. Dado que el código fuente del SGBD no está necesariamente disponible y no se puede modificar para que sea compatible con clústeres, las capacidades de gestión de datos paralelas deben implementarse mediante middleware. Este enfoque se ha adoptado con éxito en los clústeres de MySQL o PostgreSQL.

Se ha dedicado mucha investigación a aprovechar al máximo el entorno de clúster (con comunicación rápida y fiable) para mejorar el rendimiento y la disponibilidad mediante la replicación de datos. Los principales resultados de esta investigación son nuevas técnicas de replicación, balanceo de carga y procesamiento de consultas. En esta sección, presentamos estas técnicas tras introducir una arquitectura de clúster de bases de datos.

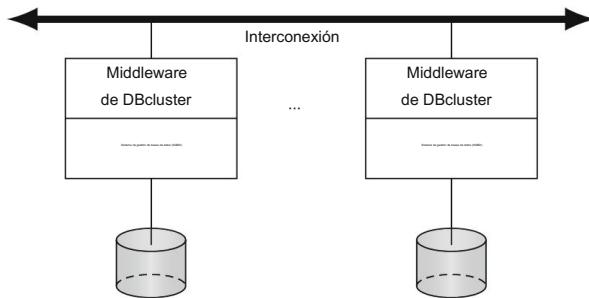


Fig. 8.19 Un clúster de base de datos sin recursos compartidos

8.7.1 Arquitectura del clúster de bases de datos

La Figura 8.19 ilustra un clúster de bases de datos con una arquitectura de no compartición. La gestión paralela de datos se realiza mediante sistemas de gestión de bases de datos (SGBD) independientes, orquestados por un middleware replicado en cada nodo. Para mejorar el rendimiento y la disponibilidad, los datos pueden replicarse en diferentes nodos utilizando el SGBD local. Las aplicaciones cliente interactúan con el middleware de forma clásica para enviar transacciones de base de datos, es decir, consultas ad hoc, transacciones o llamadas a procedimientos almacenados. Algunos nodos pueden especializarse como nodos de acceso para recibir transacciones, en cuyo caso comparten un servicio de directorio global que captura información sobre usuarios y bases de datos. El procesamiento general de una transacción a una sola base de datos es el siguiente: primero, la transacción se autentica y autoriza mediante el directorio. Si se realiza correctamente, la transacción se enruta a un SGBD en un nodo, posiblemente diferente, para su ejecución. En la Sección 8.7.4 veremos cómo este sencillo modelo puede extenderse para gestionar el procesamiento de consultas en paralelo, utilizando varios nodos para procesar una sola consulta.

Al igual que en un SGBD paralelo, el middleware del clúster de bases de datos consta de varias capas de software: balanceador de carga de transacciones, gestor de replicación, procesador de consultas y gestor de tolerancia a fallos. El balanceador de carga de transacciones activa la ejecución de transacciones en el nodo óptimo, utilizando la información de carga obtenida mediante sondeos de nodos. El "mejor" nodo se define como aquel con la menor carga de transacciones. El balanceador de carga de transacciones también garantiza que la ejecución de cada transacción cumpla con las propiedades ACID y, a continuación, envía una señal al SGBD para confirmar o cancelar la transacción. El gestor de replicación gestiona el acceso a los datos replicados y garantiza una alta consistencia, de modo que las transacciones que actualizan los datos replicados se ejecuten en el mismo orden secuencial en cada nodo. El procesador de consultas aprovecha el paralelismo entre consultas y dentro de una consulta. Con el paralelismo entre consultas, el procesador de consultas enruta cada consulta enviada a un nodo y, una vez completada, envía los resultados a la aplicación cliente. El paralelismo intraconsulta es más complejo. Dado que los SGBD de caja negra no son compatibles con clústeres, no pueden interactuar entre sí para procesar la misma consulta. Por lo tanto, el procesador de consultas controla la ejecución de la consulta, la composición del resultado final y...

Equilibrio de carga. Finalmente, el administrador de tolerancia a fallos proporciona recuperación en línea y conmutación por error.

8.7.2 Replicación

Al igual que en los SGBD distribuidos, la replicación puede utilizarse para mejorar el rendimiento y la disponibilidad. En un clúster de bases de datos, el rápido sistema de interconexión y comunicación puede aprovecharse para permitir la serialización de una sola copia, a la vez que proporciona escalabilidad (para lograr rendimiento con un gran número de nodos) y autonomía (para aprovechar un SGBD de caja negra). Un clúster proporciona un entorno estable con poca evolución de la topología (por ejemplo, debido a la adición de nodos o a fallos en los enlaces de comunicación).

Por lo tanto, es más fácil dar soporte a un sistema de comunicación grupal que gestione la comunicación confiable entre grupos de nodos. Las primitivas de comunicación grupal (véase la Sección 6.4) pueden utilizarse con técnicas de replicación ávida o diferida para lograr la diseminación atómica de información (es decir, en lugar del costoso 2PC).

Presentamos ahora otro protocolo, denominado replicación preventiva, que es perezoso y ofrece compatibilidad con la serialización y escalabilidad de una sola copia. La replicación preventiva también preserva la autonomía del DBMS. En lugar de utilizar multidifusión totalmente ordenada, utiliza multidifusión fiable FIFO, que es más sencilla y eficiente. El principio es el siguiente: cada transacción entrante T al sistema tiene una marca de tiempo cronológica $ts(T) = C$ y se multidifunde a todos los demás nodos donde hay una copia.

En cada nodo, se introduce un retraso de tiempo antes de iniciar la ejecución de T. Este retraso corresponde al límite superior del tiempo necesario para realizar la multidifusión de un mensaje (se supone un sistema sincrónico con tiempo de computación y transmisión limitado).

El problema crítico es el cálculo preciso de los límites superiores para los mensajes (es decir, el retraso). En un sistema de clúster, el límite superior se puede calcular con bastante precisión. Cuando el retraso expira, se garantiza que todas las transacciones que se hayan confirmado antes de C se recibirán y ejecutarán antes de T, siguiendo el orden de la marca de tiempo (es decir, el orden total). Por lo tanto, este enfoque evita conflictos y refuerza la consistencia en los clústeres de bases de datos. La introducción de tiempos de retraso también se ha explotado en varios protocolos de replicación centralizada diferida para sistemas distribuidos. La validación del protocolo de replicación preventiva mediante experimentos con el banco de pruebas TPC-C en un clúster de 64 nodos que ejecutan el SGBD PostgreSQL ha demostrado una excelente escalabilidad y aceleración.

8.7.3 Equilibrio de carga

En un clúster de bases de datos, la replicación ofrece buenas oportunidades para equilibrar la carga. Con la replicación diligente o preventiva (véase la sección 8.7.2), es fácil lograr el equilibrio de carga de las consultas. Dado que todas las copias son mutuamente consistentes, cualquier nodo que almacene una copia de los datos de la transacción, por ejemplo, el nodo con menos carga, puede ser seleccionado en tiempo de ejecución por un

Estrategia de balanceo de carga convencional. El balanceo de carga de transacciones también es sencillo en el caso de la replicación distribuida diferida, ya que todos los nodos maestros deben ejecutar la transacción. Sin embargo, el coste total de la ejecución de la transacción en todos los nodos puede ser elevado. Al reducir la consistencia, la replicación diferida puede reducir el coste de ejecución de las transacciones y, por lo tanto, aumentar el rendimiento tanto de las consultas como de las transacciones. Por lo tanto, según los requisitos de consistencia y rendimiento, tanto la replicación diligente como la diferida son útiles en clústeres de bases de datos.

8.7.4 Procesamiento de consultas

En un clúster de bases de datos, el procesamiento de consultas en paralelo se puede utilizar con éxito para obtener un alto rendimiento. El paralelismo entre consultas se obtiene de forma natural gracias al equilibrio de carga y la replicación, como se explicó en la sección anterior. Este paralelismo es principalmente útil para aumentar el rendimiento de las aplicaciones orientadas a transacciones y, en cierta medida, para reducir el tiempo de respuesta de las transacciones y consultas. Para las aplicaciones OLAP que suelen utilizar consultas ad hoc y que acceden a grandes cantidades de datos, el paralelismo intraconsulta es esencial para reducir aún más el tiempo de respuesta. El paralelismo intraconsulta consiste en procesar la misma consulta en diferentes particiones de las relaciones involucradas en la consulta.

Existen dos soluciones alternativas para particionar las relaciones en un clúster de bases de datos: física y virtual. El particionamiento físico define las particiones de las relaciones, esencialmente como fragmentos horizontales, y las asigna a los nodos del clúster, posiblemente con replicación. Esto se asemeja al diseño de fragmentación y asignación en bases de datos distribuidas (véase el capítulo 2), salvo que el objetivo es aumentar el paralelismo intraconsulta, no la localización de la referencia. Por lo tanto, dependiendo del tamaño de las consultas y las relaciones, el grado de particionamiento debería ser mucho más preciso. El particionamiento físico en clústeres de bases de datos para la toma de decisiones puede utilizar particiones de grano pequeño. Con una distribución uniforme de datos, se ha demostrado que esta solución produce un buen paralelismo intraconsulta y supera al paralelismo interconsulta. Sin embargo, el particionamiento físico es estático y, por lo tanto, muy sensible a las condiciones de asimetría de los datos y a la variación de los patrones de consulta, lo que puede requerir un particionado periódico.

La partición virtual evita los problemas de la partición física estática utilizando un enfoque dinámico y replicación completa (cada relación se replica en cada nodo).

En su forma más simple, denominada particionamiento virtual simple (SVP) , se generan particiones virtuales dinámicamente para cada consulta y el paralelismo intraconsulta se obtiene enviando subconsultas a diferentes particiones virtuales. Para generar las diferentes subconsultas, el procesador de consultas del clúster de bases de datos añade predicados a la consulta entrante para restringir el acceso a un subconjunto de una relación, es decir, una partición virtual. También puede reescribir la consulta para descomponerla en subconsultas equivalentes, seguida de una consulta de composición. Posteriormente, cada SGBD que recibe una subconsulta se ve obligado a procesar un subconjunto diferente de datos. Finalmente, el resultado particionado debe combinarse mediante una consulta agregada.

Ejemplo 8.6 Ilustremos SVP con la siguiente consulta Q:

```
SELECCIONAR PNO, AVG(DUR)
DE OBRAS
DONDE SUMA(DUR) > 200
GRUPO POR PNO
```

Una subconsulta genérica en una partición virtual se obtiene añadiendo a la cláusula where de Q el predicado "y PNO \geq 'P1' y PNO $<$ 'P2'". Al enlazar '[P1', 'P2]' a n rangos subsiguientes de valores PNO, se obtienen n subconsultas, cada una para un nodo diferente en una partición virtual distinta de WORKS. Por lo tanto, el grado de paralelismo entre consultas es n. Además, la operación AVG(DUR) debe reescribirse como SUM(DUR), COUNT(DUR) en la subconsulta. Finalmente, para obtener el resultado correcto para AVG(DUR), la consulta de composición debe realizar SUM(DUR)/SUM(COUNT(DUR)) sobre los n resultados parciales.

El rendimiento de la ejecución de cada subconsulta depende en gran medida de los métodos de acceso disponibles en el atributo de partición (PNO). En este ejemplo, un índice agrupado en PNO sería la mejor opción. Por lo tanto, es importante que el procesador de consultas conozca los métodos de acceso disponibles para decidir, según la consulta, qué atributo de partición utilizar.

SVP permite una gran flexibilidad para la asignación de nodos durante el procesamiento de consultas, ya que cualquier nodo puede seleccionarse para ejecutar una subconsulta. Sin embargo, no todos los tipos de consultas pueden beneficiarse de SVP y ser paralelizados. Podemos clasificar las consultas OLAP de forma que las consultas de la misma clase tengan propiedades de paralelización similares. Esta clasificación se basa en cómo se accede a las relaciones más grandes, denominadas tablas de hechos en una aplicación OLAP típica. La razón es que la partición virtual de dichas relaciones produce un mayor paralelismo intraoperador. Se identifican tres clases principales:

1. Consultas sin subconsultas que acceden a una tabla de hechos.
2. Consultas con una subconsulta que sean equivalentes a una consulta de clase 1.
3. Cualquier otra consulta.

Las consultas de Clase 2 deben reescribirse en consultas de Clase 1 para que se aplique el SVP, mientras que las consultas de Clase 3 no pueden beneficiarse del SVP.

SVP presenta algunas limitaciones. En primer lugar, determinar los mejores atributos y rangos de valores para la partición virtual puede ser difícil, ya que asumir una distribución uniforme de valores no es realista. En segundo lugar, algunos SGBD realizan escaneos completos de tablas en lugar de acceso indexado al recuperar tuplas de intervalos grandes de valores. Esto reduce las ventajas del acceso a discos en paralelo, ya que un nodo podría leer una relación completa para acceder a una partición virtual. Esto hace que SVP dependa de las capacidades de consulta del SGBD subyacente. En tercer lugar, dado que una consulta no se puede modificar externamente mientras se ejecuta, el equilibrio de carga es difícil de lograr y depende de la partición inicial.

El particionamiento virtual detallado soluciona estas limitaciones utilizando un gran número de subconsultas en lugar de una por SGBD. Trabajar con subconsultas más pequeñas evita escaneos completos de tablas y hace que el procesamiento de consultas sea menos vulnerable a las idiosincrasias del SGBD. Sin embargo, este enfoque debe estimar el tamaño de las particiones, utilizando