

Estadísticas de bases de datos y estimaciones del tiempo de procesamiento de consultas. En la práctica, estas estimaciones son difíciles de obtener con sistemas de gestión de bases de datos de caja negra.

El particionamiento virtual adaptativo (AVP) resuelve este problema ajustando dinámicamente el tamaño de las particiones, eliminando así la necesidad de estas estimaciones. AVP se ejecuta de forma independiente en cada nodo del clúster participante, evitando la comunicación entre nodos (para determinar el tamaño de las particiones). Inicialmente, cada nodo recibe un intervalo de valores con el que trabajar.

Estos intervalos se determinan exactamente igual que para SVP. Luego, cada nodo realiza los siguientes pasos:

1. Comience con un tamaño de partición muy pequeño comenzando con el primer valor de la partición recibida. intervalo.
2. Ejecute una subconsulta con este intervalo.
3. Aumente el tamaño de la partición y ejecute la subconsulta correspondiente mientras el aumento en el tiempo de ejecución sea proporcionalmente menor que el aumento en el tamaño de la partición.
4. Deje de aumentar. Se ha encontrado un tamaño estable.
5. Si hay una degradación del rendimiento, es decir, hubo ejecuciones peores consecutivas, reduzca el tamaño y vaya al Paso 2.

Comenzar con un tamaño de partición muy pequeño evita escaneos completos de tabla al inicio del proceso. Esto también evita tener que conocer el umbral a partir del cual el DBMS deja de usar índices agrupados y comienza a realizar escaneos completos de tabla. Cuando aumenta el tamaño de la partición, se monitoriza el tiempo de ejecución de las consultas, lo que permite determinar el punto a partir del cual los pasos de procesamiento de consultas independientes del tamaño de los datos no influyen demasiado en el tiempo total de ejecución. Por ejemplo, si al duplicar el tamaño de la partición se obtiene un tiempo de ejecución el doble del anterior, significa que se ha alcanzado dicho punto. Por lo tanto, el algoritmo deja de aumentar el tamaño. El rendimiento del sistema puede deteriorarse debido a fallos en la caché de datos del DBMS o al aumento de la carga general del sistema. Puede ocurrir que el tamaño utilizado sea demasiado grande y se haya beneficiado de los aciertos previos en la caché de datos. En este caso, puede ser mejor reducir el tamaño de la partición. Esto es precisamente lo que hace el paso 5: permite retroceder e inspeccionar particiones más pequeñas. Por otro lado, si el deterioro del rendimiento se debió a un aumento casual y temporal de la carga del sistema o a fallos en la caché de datos, mantener un tamaño de partición pequeño puede provocar un rendimiento deficiente. Para evitarlo, el algoritmo vuelve al paso 2 y reinicia con tamaños mayores.

AVP y otras variantes de particionamiento virtual ofrecen varias ventajas: flexibilidad para la asignación de nodos, alta disponibilidad gracias a la replicación completa y oportunidades para el balanceo de carga dinámico. Sin embargo, la replicación completa puede generar un alto costo en el uso del disco.

Para facilitar la replicación parcial, se han propuesto soluciones híbridas que combinan particionamiento físico y virtual. El diseño híbrido utiliza particionamiento físico para las relaciones más grandes e importantes y replica completamente las tablas pequeñas. De esta forma, se logra paralelismo entre consultas con menores requisitos de espacio en disco. La solución híbrida combina AVP con particionamiento físico. Resuelve el problema del uso de disco, manteniendo las ventajas de AVP: evita el escaneo completo de tablas y el balanceo de carga dinámico.

8.8 Conclusión

Los sistemas de bases de datos paralelas han aprovechado las arquitecturas multiprocesador para ofrecer alto rendimiento, alta disponibilidad, extensibilidad y escalabilidad con una buena relación calidad-precio. Además, el paralelismo es la única solución viable para soportar bases de datos y aplicaciones de gran tamaño dentro de un único sistema.

Las arquitecturas de sistemas de bases de datos paralelas se pueden clasificar como de memoria compartida, disco compartido y sin recursos compartidos. Cada arquitectura tiene sus ventajas y limitaciones. La memoria compartida se utiliza en multiprocesadores NUMA estrechamente acoplados o procesadores multinúcleo, y puede proporcionar el máximo rendimiento gracias al rápido acceso a memoria y al excelente equilibrio de carga. Sin embargo, tiene una extensibilidad y escalabilidad limitadas. El disco compartido y sin recursos compartidos se utilizan en clústeres de ordenadores, que suelen utilizar procesadores multinúcleo. Con redes de baja latencia (p. ej., Infiniband y Myrinet), pueden proporcionar un alto rendimiento y escalar a configuraciones muy grandes (con miles de nodos). Además, la capacidad RDMA de estas redes puede aprovecharse para crear clústeres NUMA rentables. El disco compartido se suele utilizar para cargas de trabajo OLTP, ya que es más sencillo y ofrece un buen equilibrio de carga.

Sin embargo, la tecnología de nada compartido sigue siendo la única opción para sistemas altamente escalables, como los que se necesitan en OLAP o big data, con la mejor relación costo/rendimiento.

Las técnicas de gestión de datos paralelos amplían las técnicas de bases de datos distribuidas.

Sin embargo, los problemas críticos para estas arquitecturas son la partición de datos, la replicación, el procesamiento de consultas paralelas, el balanceo de carga y la tolerancia a fallos. Las soluciones a estos problemas son más complejas que en los SGBD distribuidos, ya que deben escalar a un gran número de nodos. Además, los avances recientes en hardware y software, como la interconexión de baja latencia, los nodos de procesador multinúcleo, la memoria principal de gran tamaño y RDMA, ofrecen nuevas oportunidades de optimización. En particular, los algoritmos paralelos para los operadores más exigentes, como la unión y la ordenación, deben ser compatibles con NUMA.

Un clúster de bases de datos es un tipo importante de sistema de bases de datos paralelo que utiliza un SGBD de caja negra en cada nodo. Se ha dedicado mucha investigación a aprovechar al máximo el entorno estable del clúster 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.

8.9 Notas bibliográficas

La propuesta inicial de una máquina de base de datos se remonta a [Canaday et al., 1974], principalmente para abordar el "cuello de botella de E/S" [Boral y DeWitt, 1983], provocado por el alto tiempo de acceso al disco con respecto al tiempo de acceso a la memoria principal. La idea principal era acercar las funciones de la base de datos al disco. CAFS-ISP es un ejemplo temprano de dispositivo de filtrado basado en hardware [Babb, 1979] que se integraba en los controladores de disco para una búsqueda asociativa rápida. La introducción de microprocesadores de propósito general en los controladores de disco también condujo a los discos inteligentes [Keeton et al., 1998].

Los primeros sistemas de bases de datos paralelas fueron Teradata y Tandem Non-StopSQL a principios de la década de 1980. Desde entonces, todos los principales fabricantes de sistemas de gestión de bases de datos (SGBD) han lanzado una versión paralela de sus productos. Hoy en día, este campo sigue siendo objeto de intensa investigación para gestionar el big data y aprovechar las nuevas capacidades del hardware, como las interconexiones de baja latencia, los nodos de procesador multinúcleo y las memorias principales de gran capacidad.

Se ofrecen estudios exhaustivos de sistemas de bases de datos paralelas en [DeWitt y Gray 1992, Valduriez 1993, Graefe 1993]. Las arquitecturas de sistemas de bases de datos paralelas se discuten en [Bergsten et al. 1993, Stonebraker 1986, Pirahesh et al. 1990], y se comparan utilizando un modelo de simulación simple en [Breitbart y Silberschatz 1988]. Las primeras arquitecturas NUMA se describen en [Lenoski et al. 1992, Goodman y Woest 1988]. Un enfoque más reciente basado en el Acceso Directo a Memoria Remota (RDMA) se discute en [Novakovic et al. 2014, Leis et al. 2014, Barthels et al. 2015].

Ejemplos de prototipos de sistemas de bases de datos paralelos son Bubba [Boral et al. 1990], DBS3 [Bergsten et al. 1991], Gamma [DeWitt et al. 1986], Gracia [Fushimi et al. 1986], Prisma/DB [Apers et al. 1992], Volcán [Graefe 1990] y XPRS [Hong 1992].

La ubicación de datos, incluyendo la replicación, en un sistema de bases de datos paralelas se aborda en [Livny et al., 1987; Copeland et al. , 1988; Hsiao y DeWitt , 1991]. Una solución escalable es el particionamiento encadenado de Gamma [Hsiao y DeWitt, 1991], que almacena la copia principal y la de respaldo en dos nodos adyacentes. El acceso asociativo a una relación particionada mediante un índice global se propone en [Khoshafian y Valduriez, 1987].

La optimización de consultas paralelas se aborda en [Shekita et al., 1993], [Ziane et al., 1993] y [Lanzelotte et al., 1994]. Nuestro análisis del modelo de costes en la sección 8.4.2.2 se basa en [Lanzelotte et al., 1994]. Se proponen estrategias de búsqueda aleatoria en [Swami , 1989; Ioannidis y Wong , 1987]. XPRS utiliza una estrategia de optimización de dos fases [Hong y Stonebraker, 1993]. El operador de intercambio, base para la repartición paralela en el procesamiento de consultas paralelas, se propuso en el contexto del sistema de evaluación de consultas Volcano [Graefe , 1990].

Existe una extensa literatura sobre algoritmos paralelos para operadores de bases de datos, en particular sort y join. El objetivo de estos algoritmos es maximizar el grado de paralelismo, siguiendo la ley de Amdahl [Amdahl 1967] que establece que solo una parte de un algoritmo puede paralelizarse. El artículo seminal de [Bitton et al. 1983] propone y compara versiones paralelas de algoritmos de merge sort, nested loop join y sort-merge join. Valduriez y Gardarin [1984] proponen el uso de hash para algoritmos de join y semijoin paralelos. Un estudio de algoritmos de sort paralelos se puede encontrar en [Bitton et al. 1984]. La especificación de dos fases principales, build y probe, [DeWitt y Gerber 1985] ha sido útil para comprender los algoritmos de hash join paralelos. El hash join Grace [Kitsuregawa et al. 1983], el algoritmo de hash join híbrido [DeWitt et al. 1984, Shatdal et al. 1994], y la unión hash radix [Manegold et al. 2002] han sido la base de numerosas variaciones, en particular para explotar procesadores multinúcleo y NUMA [Barthels et al. 2015]. Otros algoritmos de unión importantes son la unión hash simétrica [Wilschut y Apers 1991] y la unión Ripple [Haas y Hellerstein 1999b]. En [Barthels et al. 2015], los autores demuestran que una unión hash radix puede funcionar muy bien en clústeres de gran escala sin recursos compartidos utilizando RDMA.

El algoritmo de unión de clasificación y fusión paralelo está ganando un interés renovado en el contexto de los sistemas multinúcleo y NUMA [Albutiu et al. 2012, Pasetto y Akhriev 2011].

El balanceo de carga en sistemas de bases de datos paralelas se ha estudiado extensamente tanto en el contexto de memoria y disco compartidos [Lu et al. 1991, Shekita et al. 1993] como en el de no compartir nada [Kitsuregawa y Ogawa 1990, Walton et al. 1991, DeWitt et al. 1992, Shatdal y Naughton 1993, Rahm y Marek 1995, Mehta y DeWitt 1995, Garofalakis e Ioannidis 1996]. La presentación del modelo de ejecución de procesamiento dinámico en la sección 8.5 se basa en [Bouganis et al. 1996, 1999]. El algoritmo de coincidencia de velocidad se describe en [Mehta y DeWitt 1995].

Los efectos de la distribución sesgada de datos en una ejecución paralela se presentan en [Walton et al., 1991]. En [Biscondi et al., 1996] se propone un enfoque adaptativo general para ajustar dinámicamente el grado de paralelismo mediante operadores de control. Una buena estrategia para abordar la sesga de datos consiste en utilizar múltiples algoritmos de unión, cada uno especializado para un grado diferente de sesgo, y determinar, en tiempo de ejecución, cuál es el mejor algoritmo [DeWitt et al., 1992].

El contenido de la sección 8.6 sobre tolerancia a fallos se basa en [Kemme et al. 2001, Jiménez-Peris et al. 2002, Pérez-Sorrosal et al. 2006].

El concepto de clúster de bases de datos se define en [Röhm et al. 2000, 2001].

Se proponen varios protocolos para la replicación escalable y ansiosa en clústeres de bases de datos utilizando comunicación de grupo en [Kemme y Alonso 2000b,a, Patiño-Martínez et al.

2000, Jiménez-Peris et al. 2002]. Su escalabilidad se ha estudiado analíticamente en [Jiménez-Peris et al. 2003]. La replicación parcial se estudia en [Sousa et al. 2001].

La presentación de la replicación preventiva en la Secc. 8.7.2 se basa en [Pacitti et al.

El equilibrio de carga en clústeres de bases de datos se aborda en [Milán-Franco et al. 2004 , Gançarski et al. 2007].

La mayor parte del contenido de la Sección 8.7.4 se basa en el trabajo sobre particionamiento virtual adaptativo [Lima et al., 2004] y particionamiento híbrido [Furtado et al., 2008]. El particionamiento físico en clústeres de bases de datos para el soporte de decisiones es abordado por [Stöhr et al., 2000], utilizando particiones de grano pequeño. Akal et al. [2002] proponen una clasificación de las consultas OLAP tal que las consultas de la misma clase tengan propiedades de paralelización similares.

Ceremonias

Problema 8.1 (*) Considere un clúster de discos compartidos y relaciones muy grandes que deben particionarse en varias unidades de disco. ¿Cómo adaptaría las diversas técnicas de particionamiento y replicación de la Sección 8.3 para aprovechar el uso de discos compartidos?

Analice el impacto en el rendimiento de las consultas y la tolerancia a fallas.

Problema 8.2 ()** El algoritmo hash con preservación de orden [Knuth 1973] podría utilizarse para particionar una relación en un atributo A, de modo que las tuplas en cualquier partición $i+1$ tengan valores de A mayores que los de las tuplas en la partición i . Proponga un algoritmo de ordenamiento paralelo que aproveche el algoritmo hash con preservación de orden. Analice sus ventajas y limitaciones, en comparación con el algoritmo de ordenamiento por fusión de b-way de la sección 8.4.1.1.

Problema 8.3 Considere el algoritmo de unión hash paralela de la Sección 8.4.1.2. Explique las fases de construcción y de sondeo. ¿Es el algoritmo simétrico con respecto a sus relaciones de entrada?

Problema 8.4 (*) Considere la unión de dos relaciones R y S en un clúster sin recursos compartidos. Suponga que S se partitiona mediante hash en el atributo de unión. Modifique el algoritmo de unión hash paralela de la Sección 8.4.1.2 para aprovechar este caso. Analice el coste de ejecución de este algoritmo.

Problema 8.5 (**) Considere un modelo de costos simple para comparar el rendimiento de los tres algoritmos básicos de unión paralela (unión de bucle anidado, unión de ordenación-fusión y unión hash). Se define en términos del costo total de comunicación (CCOM) y el costo de procesamiento (CPRO). Por lo tanto, el costo total de cada algoritmo es

$$\text{Costo (Alg.)} = \text{CCOM(Alg.)} + \text{CPRO(Alg.)}$$

Para simplificar, CCOM no incluye mensajes de control, necesarios para iniciar y finalizar tareas locales. Denotamos con msg(#tup) el coste de transferir un mensaje de tuplas #tup de un nodo a otro. Los costes de procesamiento (que incluyen el coste total de E/S y CPU) se basan en la función CLOC(m, n), que calcula el coste de procesamiento local para unir dos relaciones con cardinalidades m y n. Suponga que el algoritmo de unión local es el mismo para los tres algoritmos de unión en paralelo. Finalmente, suponga que la cantidad de trabajo realizado en paralelo se distribuye uniformemente entre todos los nodos asignados al operador. Proporcione las fórmulas para el coste total de cada algoritmo, suponiendo que las relaciones de entrada tienen particiones arbitrarias. Identifique las condiciones bajo las cuales se debe utilizar un algoritmo.

Problema 8.6 Considere la siguiente consulta SQL:

```
SELECCIONAR ENAME, DUR
DE EMP, ASG, PROJ DONDE
EMP.ENO=ASG.ENO
Y      ASG.PNO=PROY.PNO
Y      RESP="Gerente"
Y      PNAME="Instrumentación"
```

Proporcione cuatro posibles árboles de operadores: profundo a la derecha, profundo a la izquierda, en zigzag y arbustivo. Para Cada uno, discuta las oportunidades de paralelismo.

Problema 8.7 Considere una unión de nueve vías (se unirán diez relaciones), calcule el número de posibles árboles de profundidad derecha, profundidad izquierda y frondosos, suponiendo que cada relación puede unirse con cualquier otra. ¿Qué concluye sobre la optimización paralela?

Problema 8.8 (**) Proponga una estrategia de ubicación de datos para un clúster NUMA (usando RDMA) que maximice una combinación de paralelismo intranodo (paralelismo intraoperador dentro de nodos de memoria compartida) y paralelismo entre nodos (paralelismo interoperador a través de nodos de memoria compartida).

Problema 8.9 (**). ¿Cómo debería modificarse el modelo de ejecución de DP presentado en la Sección 8.5.4 para abordar el paralelismo entre consultas?

Problema 8.10 (**). Considere un sistema de base de datos centralizado multiusuario. Describa el cambio principal para permitir el paralelismo entre consultas desde la perspectiva del desarrollador y el administrador del sistema de base de datos. ¿Cuáles son las implicaciones para el usuario final en términos de interfaz y rendimiento?

Problema 8.11 (*). Considere la arquitectura del clúster de bases de datos de la Fig. 8.19. Suponiendo que cada nodo del clúster puede aceptar transacciones entrantes, defina el middleware del clúster de bases de datos describiendo las diferentes capas de software, sus componentes y relaciones en términos de flujo de datos y control. ¿Qué tipo de información debe compartirse entre los nodos del clúster? ¿Cómo?

Problema 8.12 (**). Analice los problemas de tolerancia a fallas para el protocolo de replicación preventiva (consulte la Sección 8.7.2).

Problema 8.13 (**). Compare el protocolo de replicación preventiva con el protocolo de replicación ansiosa (ver Cap. 6) en el contexto de un clúster de bases de datos en términos de: configuraciones de replicación admitidas, requisitos de red, consistencia, rendimiento y tolerancia a fallas.

Problema 8.14 (**). Considere dos relaciones $R(A,B,C,D,E)$ y $S(A,F,G,H)$. Suponga que existe un índice agrupado en el atributo A para cada relación. Suponiendo un clúster de base de datos con replicación completa, para cada una de las siguientes consultas, determine si se puede usar el Particionamiento Virtual para obtener paralelismo entre consultas y, de ser así, escriba la subconsulta correspondiente y la consulta de composición del resultado final. (a) SELECT

B, COUNT(C)
DESDE R
GRUPO BYB

(b) $\text{SELECCIONE C, SUMA(D), PROMEDIO(E)}$
DESDE R
DONDE B=:v1
GRUPO POR C

(c) $\text{SELECCIONAR B, SUMA(E)}$
DESDE. R, S
DONDE. RA=SA
GRUPO POR B
TENIENDO CUENTA(*) > 50

(d) $\text{SELECCIONE B, MÁX(D)}$
DESDE+. R, S
DONDE C=($\text{SELECCIONAR SUMA(G) DESDE S DONDE SA=RA}$)
GRUPO POR B

(e) $\text{SELECCIONE B, MIN(E)}$
DESDE. R
DONDE D>($\text{SELECCIONAR MAX(H) DE S DONDE G} \geq :v1$)
GRUPO POR B

Capítulo 9

Gestión de datos entre pares



En este capítulo, analizamos los problemas de gestión de datos en los sistemas de gestión de datos punto a punto (P2P) "modernos". Usamos el término "modernos" intencionalmente para diferenciarlos de los primeros sistemas P2P, comunes antes de la computación cliente-servidor. Como se indica en el capítulo 1, los primeros trabajos sobre sistemas de gestión de bases de datos distribuidos se centraron principalmente en arquitecturas P2P, donde no existía diferenciación entre la funcionalidad de cada sitio del sistema. Por lo tanto, en cierto sentido, la gestión de datos P2P es bastante antigua, si simplemente se interpreta que P2P significa que no hay "servidores" ni "clientes" identificables en el sistema. Sin embargo, los sistemas P2P "modernos" van más allá de esta simple caracterización y difieren de los sistemas antiguos que llevan el mismo nombre en varios aspectos importantes, como se menciona en el capítulo 1.

La primera diferencia es la enorme distribución de los sistemas actuales. Mientras que los primeros sistemas se centraban en unos pocos sitios (quizás decenas como máximo), los sistemas actuales consideran miles de sitios. Además, estos sitios están geográficamente muy distribuidos, con la posibilidad de que se formen agrupaciones en ciertas ubicaciones.

La segunda es la heterogeneidad inherente de cada aspecto de los sitios y su autonomía. Si bien esto siempre ha sido una preocupación de las bases de datos distribuidas, junto con la distribución masiva, la heterogeneidad y la autonomía de los sitios adquieren mayor relevancia, lo que impide considerar algunos enfoques.

La tercera gran diferencia es la considerable volatilidad de estos sistemas. Los SGBD distribuidos son entornos bien controlados, donde la adición de nuevos sitios o la eliminación de los existentes se realiza con mucho cuidado y en raras ocasiones. En los sistemas P2P modernos, los sitios son (con frecuencia) equipos individuales de usuarios, que se incorporan y abandonan el sistema P2P a voluntad, lo que dificulta considerablemente la gestión de datos.

En este capítulo, nos centramos en esta encarnación moderna de los sistemas P2P. En estos sistemas, los requisitos son los siguientes:

- Autonomía. Un par autónomo debe poder unirse o abandonar el sistema en cualquier momento sin restricciones. También debe poder controlar los datos que almacena y qué otros pares pueden almacenarlos (por ejemplo, otros pares de confianza). • Expresividad de las consultas.

El lenguaje de consulta debe permitir al usuario describir los datos deseados con el nivel de detalle adecuado. La forma más simple de consulta es la búsqueda por clave, que solo es adecuada para encontrar archivos. La búsqueda por palabras clave con clasificación de resultados es adecuada para buscar documentos, pero para datos más estructurados, se requiere un lenguaje de consulta similar a SQL.

- Eficiencia. El uso eficiente de los recursos del sistema P2P (ancho de banda, potencia de procesamiento, almacenamiento) debería resultar en un menor costo y, por lo tanto, en un mayor rendimiento de las consultas; es decir, el sistema P2P puede procesar un mayor número de consultas en un intervalo de tiempo determinado. • Calidad del servicio. Se refiere a la eficiencia del sistema percibida por el usuario, como la integridad de los resultados de las consultas, la consistencia y disponibilidad de los datos, y el tiempo de respuesta.

Tolerancia a fallos. La eficiencia y la calidad del servicio deben mantenerse a pesar de los fallos de los pares. Dada la naturaleza dinámica de los pares, que pueden abandonar o fallar en cualquier momento, es importante aprovechar adecuadamente la replicación de datos. Seguridad . La naturaleza abierta de un sistema P2P plantea serios problemas de seguridad, ya que no se puede confiar en servidores confiables. En cuanto a la gestión de datos, el principal problema de seguridad es el control de acceso, que incluye la aplicación de los derechos de propiedad intelectual sobre el contenido de los datos.

Se han desarrollado diversos usos de los sistemas P2P para compartir computación (p. ej., SETI@home), comunicación (p. ej., ICQ) o datos (p. ej., Bit-Torrent, Gnutella y Kazaa). Nuestro interés, naturalmente, se centra en los sistemas de intercambio de datos. Sistemas populares como BitTorrent, Gnutella y Kazaa presentan limitaciones considerables en cuanto a la funcionalidad de las bases de datos. En primer lugar, solo permiten compartir archivos, sin funciones sofisticadas de búsqueda/consulta basadas en contenido. En segundo lugar, son sistemas de una sola aplicación que se centran en una sola tarea, y su extensión a otras aplicaciones/funciones no es sencilla. En este capítulo, analizamos las actividades de investigación para proporcionar una funcionalidad adecuada de bases de datos en infraestructuras P2P. En este contexto, los problemas de gestión de datos que deben abordarse incluyen los siguientes:

- Ubicación de datos: los pares deben poder consultar y localizar los datos almacenados en otros pares. • Procesamiento de consultas: dada una consulta, el sistema debe poder descubrir a los pares que aportan datos relevantes y ejecutan la consulta de forma eficiente.
- Integración de datos: cuando las fuentes de datos compartidas en el sistema siguen esquemas o representaciones diferentes, los pares deberían poder acceder a esos datos, idealmente utilizando la representación de datos empleada para modelar sus propios datos. •

Consistencia de datos: si los datos se replican o almacenan en caché en el sistema, es fundamental mantener la consistencia entre estos duplicados.

La Figura 9.1 muestra una arquitectura de referencia para un par que participa en un sistema P2P de intercambio de datos. Dependiendo de la funcionalidad del sistema P2P, uno o más de los...

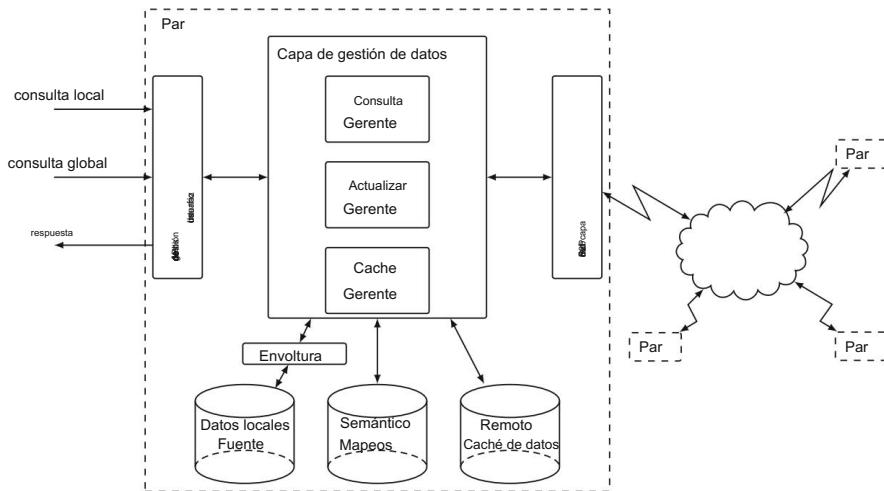


Fig. 9.1 Arquitectura de referencia de pares

Los componentes de la arquitectura de referencia pueden no existir, combinarse o implementarse por pares especializados. El aspecto clave de la arquitectura propuesta es la separación de la funcionalidad en tres componentes principales: (1) una interfaz para enviar las consultas; (2) una capa de gestión de datos que gestiona el procesamiento de consultas y la información de metadatos (p. ej., servicios de catálogo); y (3) una infraestructura P2P, compuesta por la subcapa de red P2P y la red P2P. En este capítulo, nos centramos en la capa de gestión de datos P2P y la infraestructura P2P.

Las consultas se envían mediante una interfaz de usuario o una API de gestión de datos y son gestionadas por la capa de gestión de datos. Pueden referirse a datos almacenados local o globalmente en el sistema. La solicitud de consulta es procesada por un módulo de gestión de consultas que recupera información de mapeo semántico de un repositorio cuando el sistema integra fuentes de datos heterogéneas. Este repositorio de mapeo semántico contiene metainformación que permite al gestor de consultas identificar pares en el sistema con datos relevantes para la consulta y reformular la consulta original de forma que otros pares puedan comprenderla. Algunos sistemas P2P pueden almacenar el mapeo semántico en pares especializados. En este caso, el gestor de consultas deberá contactar a estos pares especializados o transmitirles la consulta para su ejecución. Si todas las fuentes de datos del sistema siguen el mismo esquema, no se requiere el repositorio de mapeo semántico ni su funcionalidad asociada de reformulación de consultas.

Suponiendo un repositorio de mapeo semántico, el gestor de consultas invoca servicios implementados por la subcapa de la red P2P para comunicarse con los pares que participarán en la ejecución de la consulta. La ejecución real de la consulta se ve influenciada por la implementación de la infraestructura P2P. En algunos sistemas, los datos se envían al par donde se inició la consulta y luego se combinan en este par.

Otros sistemas proporcionan pares especializados para la ejecución y coordinación de consultas. En cualquier caso, los datos de resultados devueltos por los pares involucrados en la ejecución de la consulta...

Se puede almacenar en caché local para acelerar futuras ejecuciones de consultas similares. El administrador de caché mantiene la caché local de cada par. Como alternativa, el almacenamiento en caché puede ocurrir solo en pares especializados.

El gestor de consultas también se encarga de ejecutar la parte local de una consulta global cuando un par remoto solicita datos. Un contenedor puede ocultar datos, lenguaje de consulta o cualquier otra incompatibilidad entre la fuente de datos local y la capa de gestión de datos. Cuando se actualizan los datos, el gestor de actualizaciones coordina la ejecución de la actualización entre los pares que almacenan réplicas de los datos que se están actualizando.

La infraestructura de red P2P, que puede implementarse como topología de red estructurada o no estructurada, proporciona servicios de comunicación a la capa de gestión de datos.

En el resto de este capítulo, abordaremos cada componente de esta arquitectura de referencia, comenzando con los problemas de infraestructura en la Sección 9.1. Los problemas de mapeo de datos y los enfoques para abordarlos se tratan en la Sección 9.2. El procesamiento de consultas se trata en la Sección 9.3. La consistencia de los datos y los problemas de replicación se tratan en la Sección 9.4. En la Sección 9.5, presentamos Blockchain, una infraestructura P2P para registrar transacciones de forma eficiente, segura y permanente.

9.1 Infraestructura

La infraestructura de todos los sistemas P2P es una red P2P, construida sobre una red física (generalmente Internet); por lo tanto, se la conoce comúnmente como red superpuesta. La red superpuesta puede (y suele tener) una topología diferente a la de la red física, y todos los algoritmos se centran en optimizar la comunicación a través de ella (generalmente en términos de minimizar el número de "saltos" que un mensaje debe realizar desde un nodo de origen hasta un nodo de destino, ambos en la red superpuesta). La distinción entre la red superpuesta y la red física puede ser problemática, ya que dos nodos vecinos en la red superpuesta pueden, en algunos casos, estar considerablemente alejados en la red física. Por lo tanto, el coste de la comunicación dentro de la red superpuesta puede no reflejar el coste real de la comunicación en la red física.

Abordaremos esta cuestión en los momentos apropiados durante el debate sobre la infraestructura.

Las redes superpuestas pueden ser de dos tipos generales: puras e híbridas. Las redes superpuestas puras (más comúnmente conocidas como redes P2P puras) son aquellas en las que no hay diferenciación entre los nodos de la red; todos son iguales. En las redes P2P híbridas, por otro lado, algunos nodos tienen tareas específicas que realizar.

Las redes híbridas se conocen comúnmente como sistemas superpeer, ya que algunos de los pares son responsables de controlar a otros pares en su dominio. Las redes puras pueden dividirse a su vez en redes estructuradas y no estructuradas.

Las redes estructuradas controlan estrictamente la topología y el enrutamiento de los mensajes, mientras que en las redes no estructuradas cada nodo puede comunicarse directamente con sus vecinos y unirse a la red conectándose a cualquier nodo.

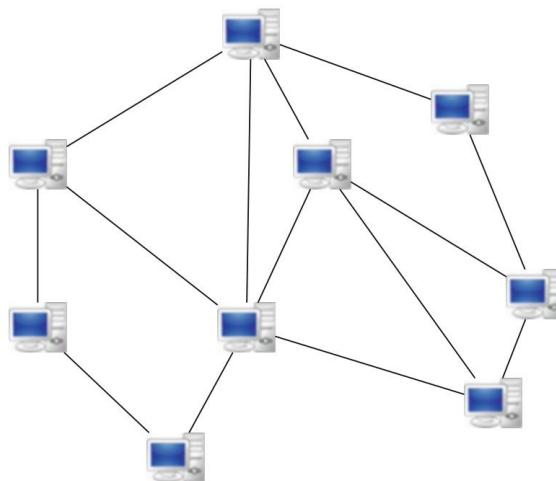


Figura 9.2 Red P2P no estructurada

9.1.1 Redes P2P no estructuradas

Las redes P2P no estructuradas son aquellas que no tienen restricciones en la ubicación de los datos en la topología superpuesta. La red superpuesta se crea de forma no determinista (ad hoc) y la ubicación de los datos es completamente independiente de la topología superpuesta. Cada par conoce a sus vecinos, pero desconoce los recursos que poseen. La Figura 9.2 muestra un ejemplo de red P2P no estructurada.

Las redes no estructuradas son los primeros ejemplos de sistemas P2P cuya funcionalidad principal es el intercambio de archivos. En estos sistemas, se comparten copias replicadas de archivos populares entre pares, sin necesidad de descargarlos desde un servidor centralizado. Ejemplos de estos sistemas son Gnutella, Freenet, Kazaa y BitTorrent.

Un aspecto fundamental en todas las redes P2P es el tipo de índice de los recursos que posee cada par, ya que esto determina cómo se buscan. Cabe destacar que la denominada "gestión de índices" en el contexto de los sistemas P2P es muy similar a la gestión de catálogos que estudiamos en el capítulo 2. Los índices son metadatos almacenados que el sistema mantiene. El contenido exacto de los metadatos varía según el sistema P2P. En general, incluye, como mínimo, información sobre los recursos y su tamaño.

Existen dos alternativas para el mantenimiento de índices: centralizada, donde un par almacena los metadatos de todo el sistema P2P, y distribuida, donde cada par mantiene los metadatos de los recursos que posee. Nuevamente, las alternativas son idénticas a las de la gestión de directorios.

El tipo de índice que admite un sistema P2P (centralizado o distribuido) influye en la forma en que se buscan los recursos. Cabe destacar que, en este punto, no nos referimos a ejecutar consultas; simplemente analizamos cómo, dado un identificador de recurso, la infraestructura P2P subyacente puede localizar el recurso relevante. En sistemas que mantienen un índice centralizado, el proceso implica consultar al par central para encontrar la ubicación.

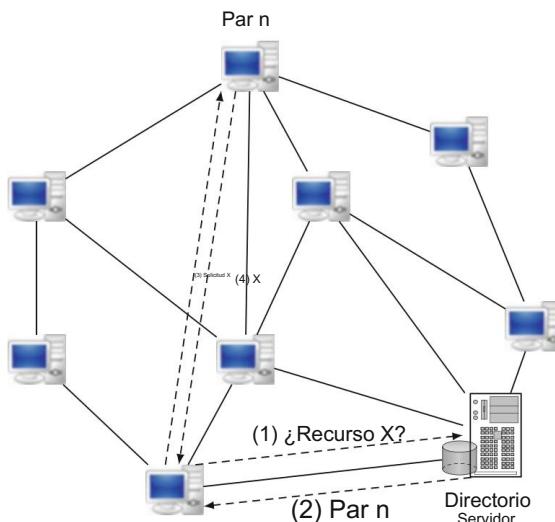


Fig. 9.3 Búsqueda en un índice centralizado. (1) Un par solicita un recurso al administrador del índice central, (2) La respuesta identifica al par con el recurso, (3) Se le solicita el recurso al par, (4) Se transfiere

del recurso, seguido de contactar directamente al par donde se encuentra (Fig. 9.3). Por lo tanto, el sistema funciona de forma similar a un cliente/servidor hasta obtener la información de índice necesaria (es decir, los metadatos), pero a partir de ese momento, la comunicación se establece únicamente entre los dos pares. Cabe destacar que el par central puede devolver un conjunto de pares que poseen el recurso y el par solicitante puede elegir uno de ellos, o bien el par central puede elegir (teniendo en cuenta, por ejemplo, la carga y las condiciones de la red) y devolver solo un par recomendado.

En sistemas que mantienen un índice distribuido, existen diversas alternativas de búsqueda. La más popular es la inundación, donde el par que busca un recurso envía la solicitud de búsqueda a todos sus vecinos en la red superpuesta. Si alguno de estos vecinos tiene el recurso, responde; de lo contrario, cada uno reenvía la solicitud a sus vecinos hasta que se encuentra el recurso o hasta que la red superpuesta se extiende por completo (Fig. 9.4).

Naturalmente, la inundación exige una gran cantidad de recursos de red y no es escalable: a medida que la red superpuesta crece, se inicia más comunicación. Esto se ha solucionado estableciendo un límite de tiempo de vida (TTL) que restringe el número de saltos que realiza un mensaje de solicitud antes de ser descartado de la red.

Sin embargo, TTL también restringe el número de nodos a los que se puede acceder.

Existen otros enfoques para abordar este problema. Un método sencillo consiste en que cada par elija un subconjunto de sus vecinos y reenvíe la solicitud solo a ellos. Existen diferentes maneras de determinar este subconjunto. Por ejemplo, se puede utilizar el concepto de recorrido aleatorio, donde cada par elige un vecino al azar.

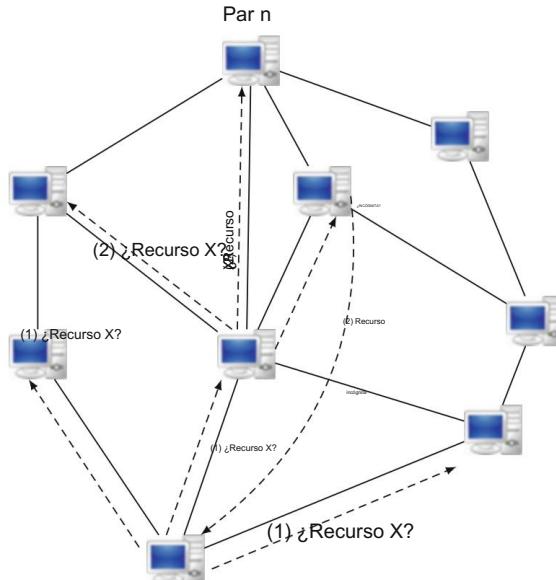


Fig. 9.4 Búsqueda en un índice descentralizado. (1) Un par envía la solicitud de recurso a todos sus vecinos, (2) Cada vecino propaga a sus vecinos si no tiene el recurso, (3) El par que tiene el recurso responde enviando el recurso.

y propaga la solicitud únicamente a este. Como alternativa, cada vecino puede mantener índices no solo para los recursos locales, sino también para los recursos de sus pares dentro de su radio y utilizar la información histórica sobre su rendimiento en las consultas de enruteamiento. Otra alternativa es utilizar índices similares basados en los recursos de cada nodo para generar una lista de vecinos con mayor probabilidad de estar en la dirección del par que contiene los recursos solicitados. Estos se conocen como índices de enruteamiento y se utilizan con mayor frecuencia en redes estructuradas, donde los analizaremos con más detalle.

Otro enfoque consiste en explotar los protocolos de chisme, también conocidos como protocolos epidémicos. El chisme se propuso inicialmente para mantener la coherencia mutua de los datos replicados mediante la difusión de actualizaciones de réplicas a todos los nodos de la red. Desde entonces, se ha utilizado con éxito en redes P2P para la difusión de datos. El chisme básico es simple. Cada nodo de la red tiene una vista completa de la red (es decir, una lista de direcciones de todos los nodos) y elige un nodo al azar para difundir la solicitud. La principal ventaja del chisme es su robustez ante fallos de nodo, ya que, con una probabilidad muy alta, la solicitud se propaga finalmente a todos los nodos de la red. Sin embargo, en redes P2P de gran tamaño, el modelo básico de chisme no es escalable, ya que mantener la vista completa de la red en cada nodo generaría un tráfico de comunicaciones muy intenso. Una solución para el chisme escalable es mantener en cada nodo solo una vista parcial de la red, por ejemplo, una lista de decenas de nodos vecinos.

Para cotillear una solicitud, un nodo elige, al azar, un nodo en su vista parcial y lo envía.

La solicitud. Además, los nodos involucrados en un chisme intercambian sus vistas parciales para reflejar los cambios de la red en sus propias vistas. Así, al actualizar continuamente sus vistas parciales, los nodos pueden autoorganizarse en superposiciones aleatorias que escalan muy bien.

El último tema que nos gustaría abordar con respecto a las redes no estructuradas es cómo los pares se unen y abandonan la red. El proceso es diferente para los enfoques de índice centralizado y distribuido. En un sistema de índice centralizado, un par que desea unirse simplemente notifica al par del índice central y le informa sobre los recursos que desea aportar al sistema P2P. En el caso de un índice distribuido, el par que se une necesita conocer a otro par en el sistema al que se "une" notificándole y recibiendo información sobre sus vecinos. En ese momento, el par forma parte del sistema y comienza a construir sus propios vecinos. Los pares que abandonan el sistema no necesitan realizar ninguna acción especial; simplemente desaparecen.

Su desaparición se detectará a tiempo y la red superpuesta se ajustará automáticamente.

9.1.2 Redes P2P estructuradas

Las redes P2P estructuradas surgieron para abordar los problemas de escalabilidad de las redes P2P no estructuradas. Logran este objetivo mediante un control estricto de la topología superpuesta y la asignación de recursos. De este modo, logran una mayor escalabilidad a costa de una menor autonomía, ya que cada par que se une a la red permite que sus recursos se asignen a ella según el método de control específico utilizado.

Al igual que en las redes P2P no estructuradas, existen dos cuestiones fundamentales que deben abordarse: cómo se indexan los recursos y cómo se buscan. El mecanismo de indexación y localización de datos más popular en las redes P2P estructuradas es una tabla hash distribuida (DHT). Los sistemas basados en DHT ofrecen dos API: `put(key, data)` y `get(key)`, donde `key` es un identificador de objeto. Cada clave (`ki`) se ha convertido en un hash para generar un identificador de par (`pi`), que almacena los datos correspondientes al contenido del objeto (Fig. 9.5).

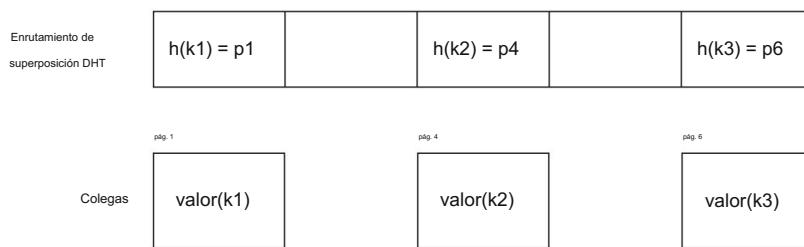


Figura 9.5 Red DHT

Un enfoque sencillo podría ser usar la URI del recurso como la dirección IP del par que lo albergaría. Sin embargo, uno de los requisitos de diseño importantes es proporcionar una distribución uniforme de los recursos en la red superpuesta, y las URI/direcciones IP no ofrecen suficiente flexibilidad. Por consiguiente, se utilizan técnicas de hash consistentes que proporcionan un hash uniforme de los valores para distribuir los datos de forma uniforme en la red superpuesta. Si bien se pueden emplear diversas funciones hash para generar asignaciones de direcciones virtuales para el recurso, SHA-1 se ha convertido en la función hash base¹ más aceptada, que ofrece uniformidad y seguridad (al garantizar la integridad de los datos de las claves). El diseño real de la función hash puede depender de la implementación, por lo que no se abordará este tema en detalle.

La búsqueda (comúnmente llamada "búsqueda") en una red P2P estructurada basada en DHT también implica la función hash: se aplica un hash a la clave del recurso para obtener el ID del par en la red superpuesta responsable de dicha clave. A continuación, se inicia la búsqueda en la red superpuesta para localizar el nodo objetivo en cuestión. Esto se conoce como protocolo de enrutamiento, difiere entre las distintas implementaciones y está estrechamente relacionado con la estructura de superposición utilizada. A continuación, analizaremos un ejemplo de enfoque.

Si bien todos los protocolos de enrutamiento buscan ofrecer búsquedas eficientes, también intentan minimizar la información de enrutamiento (también denominada estado de enrutamiento) que debe mantenerse en una tabla de enrutamiento en cada par de la superposición. Esta información varía entre los distintos protocolos de enrutamiento y estructuras de superposición, pero debe proporcionar suficiente información de tipo directorio para enrutar las solicitudes put y get al par correspondiente en la superposición. Todas las implementaciones de tablas de enrutamiento requieren el uso de algoritmos de mantenimiento para mantener el estado de enrutamiento actualizado y consistente.

A diferencia de los enruteadores en Internet, que también mantienen bases de datos de enrutamiento, los sistemas P2P presentan un mayor desafío, ya que se caracterizan por una alta volatilidad de nodos y enlaces de red poco confiables. Dado que los DHT también deben admitir una recuperación perfecta (es decir, deben encontrarse todos los recursos accesibles mediante una clave dada), la consistencia del estado de enrutamiento se convierte en un desafío clave. Por lo tanto, es esencial mantener un estado de enrutamiento consistente ante búsquedas concurrentes y durante períodos de alta volatilidad de la red.

Se han propuesto numerosas superposiciones basadas en DHT. Estas se pueden clasificar según su geometría y algoritmo de enrutamiento. La geometría de enrutamiento define esencialmente la forma en que se organizan los vecinos y las rutas. El algoritmo de enrutamiento corresponde al protocolo de enrutamiento descrito anteriormente y se define como la forma en que se eligen los siguientes saltos/rutas en una geometría de enrutamiento dada.

Las superposiciones basadas en DHT existentes más importantes se pueden clasificar de la siguiente manera:

- Árbol. En el enfoque trie, los nodos hoja corresponden a los identificadores de nodo que almacenan las claves que se buscarán. La altura del trie es $\log n$, donde n es el número de nodos del trie. La búsqueda se realiza desde la raíz hasta las hojas.

¹Una función hash base se define como una función que se utiliza como base para el diseño de otra función hash.

Se realiza una coincidencia del prefijo más largo en cada nodo intermedio hasta encontrar el nodo objetivo. Por lo tanto, en este caso, la coincidencia puede considerarse como la corrección de los valores de bits de izquierda a derecha en cada salto sucesivo del trie. Una implementación popular de DHT que entra en esta categoría es Tapestry, que utiliza enrutamiento sustituto para reenviar las solicitudes en cada nodo al dígito más cercano en la tabla de enrutamiento. El enrutamiento sustituto se define como el enrutamiento al dígito más cercano cuando no se puede encontrar una coincidencia exacta en el prefijo más largo. En Tapestry, cada identificador único se asocia a un nodo que es la raíz de un trie de expansión único que se utiliza para enrutar los mensajes para el identificador dado. Por lo tanto, las búsquedas proceden desde la base del trie de expansión hasta el nodo raíz del identificador. Si bien esto es algo diferente de las estructuras de trie tradicionales, la geometría de enrutamiento de Tapestry está muy asociada a una estructura de trie y la clasificamos como tal.

En las estructuras trie, un nodo del sistema tiene $2i-1$ nodos para elegir como su vecino del subárbol con el que tiene $\log(n - i)$ bits de prefijo en común.

El número de vecinos potenciales aumenta exponencialmente a medida que avanzamos en el trie. Por lo tanto, en total hay $n \log n/2$ tablas de enrutamiento posibles por nodo (tenga en cuenta, sin embargo, que solo se puede seleccionar una tabla de enrutamiento de este tipo para un nodo). Por lo tanto, la geometría trie tiene buenas características de selección de vecinos que le proporcionarían tolerancia a fallos. Sin embargo, el enrutamiento solo puede realizarse a través de un nodo vecino cuando se envía a un destino particular. En consecuencia, los DHT estructurados en trie no proporcionan ninguna flexibilidad en la selección de rutas. •

Hipercubo. La geometría de enrutamiento del hipercubo se basa en un espacio de coordenadas cartesianas d-dimensional que se divide en un conjunto individual de zonas de modo que cada nodo mantiene una zona separada del espacio de coordenadas. Un ejemplo de DHT basado en hipercubo es la Red Direccional por Contenido (CAN). El número de vecinos que un nodo puede tener en un espacio de coordenadas d-dimensional es $2d$ (para fines de discusión, consideraremos $d = \log n$). Si consideramos que cada coordenada representa un conjunto de bits, entonces cada identificador de nodo puede representarse como una cadena de bits de longitud $\log n$. De esta manera, la geometría del hipercubo es muy similar al trie, ya que también simplemente fija los bits en cada salto para llegar al destino. Sin embargo, en el hipercubo, dado que los bits de los nodos vecinos solo difieren en exactamente un bit, cada nodo de reenvío necesita modificar solo un bit en la cadena de bits, lo que puede hacerse en cualquier orden. Por lo tanto, si consideramos la corrección de la cadena de bits, la primera corrección se puede aplicar a cualquier $\log n$ nodos, la siguiente corrección se puede aplicar a cualquier $(\log n) - 1$ nodos, etc. Por lo tanto, tenemos $(\log n)!$ posibles rutas entre nodos, lo que proporciona una alta flexibilidad de ruta en la geometría de enrutamiento del hipercubo. Sin embargo, un nodo en el espacio de coordenadas no tiene ninguna opción sobre las coordenadas de sus vecinos ya que las zonas de coordenadas adyacentes en el espacio de coordenadas no pueden cambiar. Por lo tanto, los hipercubos tienen poca flexibilidad de selección de vecinos. • Anillo. La geometría del anillo se representa como un espacio de identificadores circular unidimensional donde los nodos se colocan en diferentes ubicaciones en el círculo. La distancia entre dos nodos cualesquiera en el círculo es la diferencia de identificadores numéricos (en el sentido de las agujas del reloj) alrededor del círculo. Dado que el círculo es unidimensional, los identificadores de datos se pueden representar como dígitos decimales individuales (representados como cadenas de bits binarios) que se asignan a un nodo que está más cerca en el espacio de identificadores al nodo dado.

Dígito decimal. Chord es un ejemplo popular de geometría de anillo. Específicamente, en Chord, un nodo cuyo identificador es a mantiene información sobre $\log n$ de otros vecinos en el anillo, donde el i -ésimo vecino es el nodo más cercano a $a + 2i - 1$ en el círculo. Mediante estos enlaces (llamados dedos), Chord puede enrutar a cualquier otro nodo en $\log n$ saltos.

Un análisis cuidadoso de la estructura de Chord revela que un nodo no necesariamente necesita mantener como vecino al nodo más cercano a $a + 2i - 1$. De hecho, aún puede mantener el límite superior de búsqueda de $\log n$ si se elige cualquier nodo del rango $[(a + 2i - 1), (a + 2i)]$. Por lo tanto, en términos de flexibilidad de ruta, puede seleccionar entre $n \log n/2$ tablas de enrutamiento para cada nodo. Esto proporciona una gran flexibilidad en la selección de vecinos. Además, para enrutar a cualquier nodo, el primer salto tiene $\log n$ vecinos que pueden enrutar la búsqueda al destino y el siguiente nodo tiene $(\log n) - 1$ nodos, y así sucesivamente. Por lo tanto, normalmente hay $(\log n)!$ rutas posibles al destino. En consecuencia, la geometría de anillo también proporciona una buena flexibilidad en la selección de rutas.

Además de estas geometrías más populares, ha habido muchas otras DHT-basadas en superposiciones estructuradas que utilizan diferentes topologías.

Las superposiciones basadas en DHT son eficientes, ya que garantizan la búsqueda del nodo donde colocar o la búsqueda de los datos en $\log n$ saltos, donde n es el número de nodos del sistema. Sin embargo, presentan varios problemas, especialmente desde la perspectiva de la gestión de datos. Uno de los problemas con las DHT que emplean funciones hash consistentes para una mejor distribución de recursos es que dos pares que son "vecinos" en la red superpuesta debido a la proximidad de sus valores hash pueden estar geográficamente muy separados en la red real. Por lo tanto, la comunicación con un vecino en la red superpuesta puede generar altos retrasos de transmisión en la red real. Se han realizado estudios para superar esta dificultad mediante el diseño de funciones hash que tienen en cuenta la proximidad o la localidad. Otra dificultad es que no ofrecen flexibilidad en la ubicación de los datos: cada dato debe colocarse en el nodo determinado por la función hash. Por lo tanto, si hay nodos P2P que aportan sus propios datos, deben estar dispuestos a que estos se transfieran a otros nodos. Esto es problemático desde la perspectiva de la autonomía de los nodos. La tercera dificultad radica en la dificultad de ejecutar consultas de rango en arquitecturas basadas en DHT, ya que, como es bien sabido, es difícil ejecutar consultas de rango en índices hash. Se han realizado estudios para superar esta dificultad, que analizaremos más adelante.

Estas preocupaciones han impulsado el desarrollo de superposiciones estructuradas que no utilizan DHT para el enrute. En estos sistemas, los pares se asignan al espacio de datos en lugar del espacio de claves hash. Existen múltiples maneras de particionar el espacio de datos entre varios pares.

- Estructura jerárquica. Muchos sistemas emplean estructuras jerárquicas superpuestas, como trie, árboles balanceados, árboles balanceados aleatorios (p. ej., lista de omisión) y otros. Específicamente, PHT y P-Grid emplean una estructura binaria de trie, donde los pares cuyos datos comparten prefijos comunes se agrupan en ramas comunes. Los árboles balanceados también se utilizan ampliamente debido a su eficiencia de enrute garantizada (la "longitud de salto" esperada entre pares arbitrarios es proporcional a la altura del trie).

Por ejemplo, BATON, VBI-tree y BATON* emplean una estructura de trie balanceada de k vías para gestionar pares, y los datos se reparten equitativamente entre los pares a nivel de hoja. En comparación, P-Tree utiliza una estructura de árbol B con mayor flexibilidad para los cambios estructurales de los tries. SkipNet y Skip Graph se basan en la lista de omisión y vinculan pares según una estructura de trie balanceada aleatoria, donde el orden de los nodos se determina por los valores de los datos de cada nodo.

Curva de relleno. Esta arquitectura se utiliza habitualmente para linealizar datos ordenados en un espacio de datos multidimensional. Los pares se organizan a lo largo de la curva de relleno (p. ej., la curva de Hilbert) para que sea posible el recorrido ordenado de los pares según el orden de los datos.

- Estructura hiperrectangular. En estos sistemas, cada dimensión del hiperrectángulo corresponde a un atributo de los datos según el cual se desea una organización. Los pares se distribuyen en el espacio de datos de forma uniforme o según su ubicación (p. ej., mediante la relación de intersección de datos). El espacio hiperrectangular se divide entonces por pares según sus posiciones geométricas, y los pares vecinos se interconectan para formar la red superpuesta.

9.1.3 Redes P2P superpeer

Los sistemas P2P de superpeers son un híbrido entre los sistemas P2P puros y las arquitecturas cliente-servidor tradicionales. Se asemejan a estas arquitecturas en que no todos los pares son iguales; algunos pares (denominados superpeers) actúan como servidores dedicados para otros pares y pueden realizar funciones complejas como indexación, procesamiento de consultas, control de acceso y gestión de metadatos. Si solo hay un superpeer en el sistema, se reduce a la arquitectura cliente-servidor. Sin embargo, se consideran sistemas P2P, ya que la organización de los superpeers sigue una estructura P2P y pueden comunicarse entre sí de forma sofisticada. Por lo tanto, a diferencia de los sistemas cliente-servidor, la información global no está necesariamente centralizada y puede particionarse o replicarse entre superpeers.

En una red de superpeers, el par solicitante envía la solicitud, que puede expresarse en un lenguaje de alto nivel, a su superpeer responsable. Este puede entonces encontrar los pares relevantes directamente a través de su índice o indirectamente utilizando a sus superpeers vecinos. Más precisamente, la búsqueda de un recurso se realiza de la siguiente manera (véase la Fig. 9.6):

1. Un par, digamos el Par 1, solicita un recurso enviando una solicitud a su superpar.
2. Si el recurso existe en uno de los pares controlados por este superpeer, este notifica al par 1, y ambos se comunican para recuperarlo. De lo contrario, el superpeer envía la solicitud a los demás superpeers.
3. Si el recurso no existe en uno de los pares controlados por este superpeer, este solicita la asistencia de los demás. El superpeer del nodo que contiene el recurso (por ejemplo, el par n) responde al superpeer solicitante.

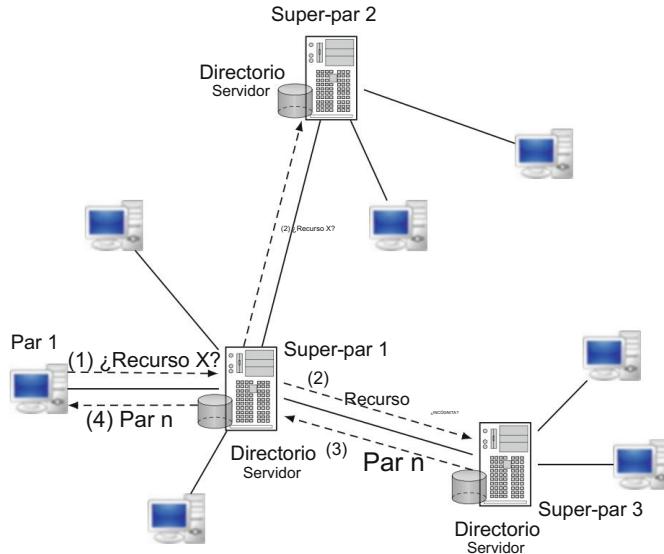


Fig. 9.6 Búsqueda en un sistema superpeer. (1) Un par envía la solicitud de recurso a todos sus superpeer, (2) El superpeer envía la solicitud a otros superpeer si es necesario, (3) El superpeer, uno de cuyos pares tiene el recurso, responde indicando ese par, (4) El superpeer notifica al par original

4. La identidad del par n se envía al par 1, después de lo cual los dos pares pueden comunicarse directamente para recuperar el recurso.

Las principales ventajas de las redes superpeer son la eficiencia y la calidad del servicio (p. ej., integridad de los resultados de las consultas, tiempo de respuesta de las consultas). El tiempo necesario para encontrar datos accediendo directamente a los índices en un superpeer es muy pequeño en comparación con la inundación. Además, las redes superpeer explotan y se aprovechan de las diferentes capacidades de los pares en términos de potencia de CPU, ancho de banda o capacidad de almacenamiento, ya que los superpeers asumen una gran parte de toda la carga de la red. El control de acceso también se puede aplicar mejor, ya que la información de directorio y seguridad se puede mantener en los superpeers. Sin embargo, la autonomía es restringida, ya que los pares no pueden iniciar sesión libremente en ningún superpeer. La tolerancia a fallos suele ser menor, ya que los superpeers son puntos únicos de fallo para sus subpeers (el reemplazo dinámico de superpeers puede aliviar este problema).

Algunos ejemplos de redes superpeer son Edutella y JXTA.

Requisitos	Superpar estructurado	no estructurado	
Autonomía	Bajo	Bajo	Moderado
Alta expresividad de la consulta			
Eficiencia	Bajo	Alto	Alto
Catálogo de servicios	Bajo	Alto	Alto
Tolerancia a fallos	Alto	Alto	Bajo
Seguridad	Bajo	Bajo	Alto

Fig. 9.7 Comparación de enfoques

9.1.4 Comparación de redes P2P

La figura 9.7 resume cómo se cumplen los requisitos para la gestión de datos (autonomía, expresividad de la consulta, eficiencia, calidad del servicio, tolerancia a fallos y seguridad) se alcanzan posiblemente mediante las tres clases principales de redes P2P. Esto es un cálculo aproximado. comparación para comprender los méritos respectivos de cada clase. Obviamente, hay Hay margen de mejora en cada clase de redes P2P. Por ejemplo, la tolerancia a fallos. Se puede mejorar en sistemas superpares confiando en la replicación y la commutación por error. Técnicas. La expresividad de las consultas se puede mejorar al admitir consultas más complejas. consultas sobre redes estructuradas.

9.2 Mapeo de esquemas en sistemas P2P

Discutimos la importancia y las técnicas para diseñar bases de datos. sistemas de integración en el Cap. 7. Surgen problemas similares en los sistemas P2P de intercambio de datos. Debido a las características específicas de los sistemas P2P, por ejemplo, la dinámica y la autonomía naturaleza de los pares, los enfoques que se basan en esquemas globales centralizados ya no aplicar. El problema principal es soportar el mapeo de esquemas descentralizados para que una consulta Lo expresado en el esquema de un par se puede reformular como una consulta en el esquema de otro par. esquema. Los enfoques que utilizan los sistemas P2P para definir y crear Las asignaciones entre esquemas de pares se pueden clasificar de la siguiente manera: esquema por pares mapeo, mapeo basado en técnicas de aprendizaje automático, acuerdo común mapeo y mapeo de esquemas usando técnicas de recuperación de información (IR).

9.2.1 Mapeo de esquemas por pares

En este enfoque, cada usuario define la asignación entre el esquema local y el esquema de cualquier otro par que contenga datos de interés. Confirmando en el transitividad de las asignaciones definidas, el sistema intenta extraer asignaciones entre esquemas que no tienen un mapeo definido.

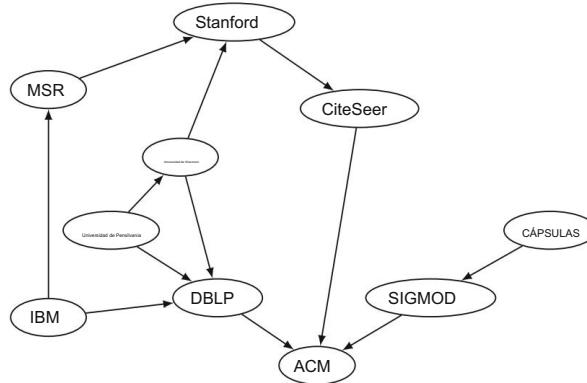


Fig. 9.8 Un ejemplo de mapeo de esquemas por pares en piazza

Piazza sigue este enfoque (véase la Fig. 9.8). Los datos se comparten como documentos XML, y cada par tiene un esquema que define la terminología y las restricciones estructurales del par. Cuando un nuevo par (con un nuevo esquema) se une al sistema por primera vez, asigna su esquema al de otros pares del sistema.

Cada definición de mapeo comienza con una plantilla XML que coincide con una ruta o subárbol de una instancia del esquema de destino. Los elementos de la plantilla pueden anotarse con expresiones de consulta que vinculan variables a nodos XML en el origen.

El Modelo Relacional Local (LRM) es otro ejemplo que sigue este enfoque. El LRM asume que los pares poseen bases de datos relacionales y que cada par conoce un conjunto de pares con los que puede intercambiar datos y servicios. Este conjunto de pares se denomina "conocidos del par". Cada par debe definir dependencias semánticas y reglas de traducción entre sus datos y los datos compartidos por cada uno de sus conocidos. Las asignaciones definidas forman una red semántica, que se utiliza para la reformulación de consultas en el sistema P2P.

Hyperion generaliza este enfoque para gestionar pares autónomos que se conocen en tiempo de ejecución, utilizando tablas de mapeo para definir correspondencias de valores entre bases de datos heterogéneas. Los pares realizan consultas locales y procesan actualizaciones, y también propagan consultas y actualizaciones a sus pares conocidos.

PGrid también asume la existencia de mapeos por pares entre pares, inicialmente construidos por expertos cualificados. Basándose en la transitividad de estos mapeos y utilizando un algoritmo de cotilleo, PGrid extrae nuevos mapeos que relacionan los esquemas de los pares entre los cuales no existe un mapeo de esquemas predefinido.

9.2.2 Mapeo basado en técnicas de aprendizaje automático

Este enfoque se utiliza generalmente cuando los datos compartidos se definen con base en ontologías y taxonomías, como las propuestas para la web semántica. Utiliza aprendizaje automático.

técnicas para extraer automáticamente las asignaciones entre los esquemas compartidos. Las asignaciones extraídas se almacenan en la red para procesar consultas futuras. GLUE utiliza este enfoque. Dadas dos ontologías, para cada concepto de una, GLUE encuentra el concepto más similar de la otra. Proporciona definiciones probabilísticas bien fundamentadas para diversas medidas prácticas de similitud y utiliza múltiples estrategias de aprendizaje, cada una de las cuales aprovecha un tipo diferente de información, ya sea en las instancias de datos o en la estructura taxonómica de las ontologías. Para mejorar aún más la precisión de las asignaciones, GLUE incorpora conocimiento de sentido común y restricciones de dominio en el proceso de asignación de esquemas. La idea básica es proporcionar clasificadores para los conceptos. Para determinar la similitud entre dos conceptos X e Y, los datos del concepto Y se clasifican utilizando el clasificador de X y viceversa. El número de valores que se pueden clasificar con éxito en X e Y representa la similitud entre X e Y.

9.2.3 Mapeo de acuerdos comunes

En este enfoque, los pares con un interés común acuerdan una descripción de esquema común para compartir datos. Este esquema común suele ser preparado y mantenido por usuarios expertos. El sistema P2P APPA asume que los pares que desean cooperar, por ejemplo, durante un experimento, acuerdan una Descripción de Esquema Común (CSD). Dado un CSD, se puede especificar un esquema de par mediante vistas. Esto es similar al enfoque LAV en sistemas de integración de datos, excepto que las consultas a un par se expresan en términos de las vistas locales, no del CSD. Otra diferencia entre este enfoque y el LAV es que el CSD no es un esquema global; es decir, es común a un conjunto limitado de pares con un interés común (véase la figura 9.9).

Por lo tanto, el CSD no presenta problemas de escalabilidad. Cuando un par decide compartir datos, necesita mapear su esquema local al CSD.

Ejemplo 9.1 Dadas dos definiciones de relación CSD R1 y R2, un ejemplo de mapeo de pares en el par p es

$$p : R(A, B, D) \quad csd : R1(A, B, C), csd : R2(C, D, E)$$

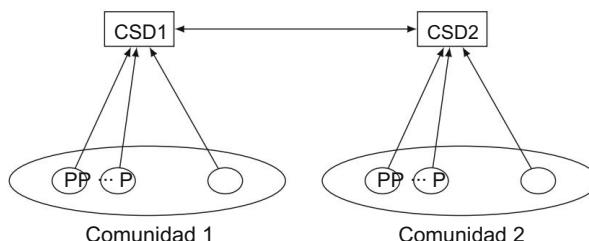


Figura 9.9 Mapeo del esquema de acuerdo común en APPA

En este ejemplo, la relación $R(A, B, D)$ compartida por el par p se asigna a las relaciones $R1(A, B, C)$ y $R2(C, D, E)$, ambas involucradas en el CSD. En APPA, las asignaciones entre el CSD y el esquema local de cada par se almacenan localmente en el par. Dada una consulta Q en el esquema local, el par reformula Q a una consulta en el CSD utilizando las asignaciones almacenadas localmente.

9.2.4 Mapeo de esquemas mediante técnicas IR

Este enfoque extrae las asignaciones de esquemas durante la ejecución de la consulta mediante técnicas de IR, explorando las descripciones de esquema proporcionadas por los usuarios. PeerDB sigue este enfoque para el procesamiento de consultas en redes P2P no estructuradas. Para cada relación compartida por un par, la descripción de la relación y sus atributos se conservan en ese par. Los usuarios proporcionan las descripciones al crear las relaciones y sirven como sinónimos de los nombres y atributos de las relaciones. Cuando se emite una consulta, se genera una solicitud para encontrar posibles coincidencias y se envía a los pares que devuelven los metadatos correspondientes. Al comparar palabras clave con los metadatos de las relaciones, PeerDB puede encontrar relaciones potencialmente similares a las relaciones de la consulta. Las relaciones encontradas se presentan al emisor de la consulta, quien decide si procede o no con la ejecución de la consulta en el par remoto propietario de las relaciones.

Edutella también sigue este enfoque para el mapeo de esquemas en redes superpeer. Los recursos en Edutella se describen mediante el modelo de metadatos RDF, y las descripciones se almacenan en superpeers. Cuando un usuario realiza una consulta a un par p , esta se envía al superpeer de p , donde se exploran las descripciones de esquema almacenadas y se devuelven las direcciones de los pares relevantes. Si el superpeer no encuentra pares relevantes, envía la consulta a otros superpeers para que los busquen explorando sus descripciones de esquema almacenadas. Para explorar los esquemas almacenados, los superpeers utilizan el lenguaje de consulta RDF-QEL, basado en la semántica de Datalog y, por lo tanto, compatible con todos los lenguajes de consulta existentes, y admite funcionalidades que amplían los lenguajes de consulta relacionales habituales.

9.3 Consultas a través de sistemas P2P

Las redes P2P proporcionan técnicas básicas para enrutar consultas a pares relevantes, lo cual es suficiente para realizar consultas simples de coincidencia exacta. Por ejemplo, como se mencionó anteriormente, una DHT proporciona un mecanismo básico para buscar datos eficientemente basándose en un valor clave. Sin embargo, realizar consultas más complejas en sistemas P2P, en particular en DHT, es difícil y ha sido objeto de mucha investigación reciente. Los principales tipos de consultas complejas útiles en sistemas P2P son las consultas top-k, las consultas de unión y las consultas de rango. En esta sección, analizamos las técnicas para procesarlas.

9.3.1 Consultas Top-k

Las consultas top-k se han utilizado en diversos ámbitos, como la monitorización de redes y sistemas, la recuperación de información y las bases de datos multimedia. Con una consulta top-k, el usuario solicita al sistema las k respuestas más relevantes. El grado de relevancia (puntuación) de las respuestas a la consulta se determina mediante una función de puntuación.

Las consultas top-k son muy útiles para la gestión de datos en sistemas P2P, en particular cuando el conjunto de respuestas completo es muy grande.

Ejemplo 9.2 Considere un sistema P2P con médicos que desean compartir datos (restringidos) de pacientes para un estudio epidemiológico. Supongamos que todos los médicos acordaron una descripción común del paciente en formato relacional. Entonces, un médico podría querer enviar la siguiente consulta para obtener las 10 respuestas principales, clasificadas según una función de puntuación basada en la altura y el peso:

```

SELECCIONAR *
DEL Paciente P
DONDE P.enfermedad = "diabetes"
Y      P.altura < 170 P.peso
Y      > 160 ORDENAR
POR función-de-puntuación(altura,peso)
PARE DESPUÉS DE 10

```

La función de puntuación especifica en qué medida cada elemento de datos coincide con las condiciones. Por ejemplo, en la consulta anterior, la función de puntuación podría calcular las diez personas con mayor sobrepeso.

La ejecución eficiente de consultas top-k en sistemas P2P es difícil debido a la escala de la red. En esta sección, primero analizamos las técnicas más eficientes propuestas para el procesamiento de consultas top-k en sistemas distribuidos. A continuación, presentamos las técnicas propuestas para sistemas P2P.

9.3.1.1 Técnicas básicas

Un algoritmo eficiente para el procesamiento de consultas top-k en sistemas centralizados y distribuidos es el algoritmo de umbral (TA). El AT se aplica a consultas donde la función de puntuación es monótona; es decir, cualquier aumento en el valor de la entrada no disminuye el valor de la salida. Muchas de las funciones de agregación populares, como Mín, Máx y Promedio, son monótonas. El AT ha sido la base de varios algoritmos, que se analizan en esta sección.

Algoritmo de umbral (AT)

TA asume un modelo basado en listas de elementos de datos ordenados por sus puntuaciones locales. El modelo es el siguiente. Supongamos que tenemos m listas de n elementos de datos tales que cada dato. El elemento tiene una puntuación local en cada lista y las listas se ordenan según la puntuación local. Las puntuaciones de sus elementos de datos. Además, cada elemento de datos tiene una puntuación general que es calculado en función de sus puntuaciones locales en todas las listas utilizando una función de puntuación dada. Para Por ejemplo, considere la base de datos (es decir, tres listas ordenadas) de la figura 9.10. Suponiendo que La función de puntuación calcula la suma de las puntuaciones locales del mismo elemento de datos en todos las listas, la puntuación global del elemento d1 es $30 + 21 + 14 = 65$.

Entonces, el problema del procesamiento de la consulta top-k es encontrar los k elementos de datos cuyos Las puntuaciones generales son las más altas. Este modelo de problema es simple y general. Supongamos Queremos encontrar las k tuplas principales en una tabla relacional según algún sistema de puntuación. función sobre sus atributos. Para responder a esta pregunta, basta con tener una función ordenada Lista (indexada) de los valores de cada atributo involucrado en la función de puntuación, y devuelve las k tuplas cuyas puntuaciones generales en las listas son las más altas. Como otra Por ejemplo, supongamos que queremos encontrar los k documentos principales cuyo rango agregado es el más alto con respecto a un conjunto dado de palabras clave. Para responder a esta pregunta, La solución es tener, para cada palabra clave, una lista ordenada de documentos y devolver los k documentos cuyo rango agregado en todas las listas es el más alto.

TA considera dos modos de acceso a una lista ordenada. El primer modo es ordenado (o Acceso secuencial) que accede a cada elemento de datos en su orden de aparición en el lista. El segundo modo es el acceso aleatorio mediante el cual se obtiene un elemento de datos determinado de la lista buscado directamente, por ejemplo, usando un índice en el ID del elemento.

Dadas m listas ordenadas de n elementos de datos, TA (ver Algoritmo 9.1) recorre las listas ordenadas. listas en paralelo y, para cada elemento de datos, recupera sus puntuaciones locales en todas las listas a través de Acceso aleatorio y calcula la puntuación general. También mantiene un conjunto Y , los datos k

Posición	Lista 1		Lista 2		Lista 3	
	Datos locales	Puntuación del artículo	Datos locales	Puntuación del artículo	Datos locales	Puntuación del artículo
1	d1	30	d2	28	d3	30
2	d4	28	d6	27	d5	29
3	d9	27	d7	25	d8	28
4	d3	26	d5	24	d4	25
5	d7	25	d9	23	d2	24
6	d8	23	d1	21	d6	19
7	d5	17	d8	20	d13	15
8	d6	14	d3	14	d1	14
9	d2	11	d4	13	d9	12
10	d11	10	d14	12	d7	11
...

Fig. 9.10 Ejemplo de base de datos con 3 listas ordenadas

Algoritmo 9.1: Algoritmo de umbral (TA)

Entrada: L₁, L₂, ..., L_m: m listas ordenadas de n elementos de datos f : función de puntuación Salida: Y : lista de los k elementos de datos principales

comienza con j ← 1 umbral ←

```

1 min_overall_score ← 0 mientras j = n + 1 y min_overall_score < umbral hacer
    {Realice acceso ordenado en paralelo a cada una de las m listas
    ordenadas} para i desde 1 hasta m en paralelo
        {Procesar cada elemento de datos en la
        posición j} para cada elemento de datos d en la
            | posición j en Li hacer {acceder a los puntajes locales de d en las otras listas a través
            | de acceso aleatorio} puntuación_general(d) ← f (puntajes
            de d en cada Li) fin para
    fin para
    Y ← k elementos de datos con la puntuación más alta
    hasta el momento min_overall_score ← puntuación general más pequeña de
    elementos de datos en Y umbral ← f (puntuaciones locales en la
    posición j en
    cada Li) j ← j + 1 fin mientras
fin

```

Elementos cuyas puntuaciones generales son las más altas hasta el momento. El mecanismo de detención del TA utiliza un umbral que se calcula utilizando las últimas puntuaciones locales observadas bajo acceso ordenado en las listas. Por ejemplo, considere la base de datos de la Fig. 9.10. En la posición 1 para todas las listas (es decir, cuando solo se han visto los primeros elementos de datos bajo acceso ordenado), asumiendo que la función de puntuación es la suma de las puntuaciones, el umbral es 30 + 28 + 30. En la posición 2, es 84. Dado que los elementos de datos se ordenan en las listas en orden decreciente de puntuación local, el umbral disminuye a medida que se avanza en la lista. Este proceso continúa hasta que se encuentran k elementos de datos cuyas puntuaciones generales sean mayores que un umbral.

Ejemplo 9.3 Considere nuevamente la base de datos (es decir, tres listas ordenadas) que se muestra en la Figura 9.10. Supongamos una consulta Q de los 3 primeros resultados (es decir, k = 3) y que la función de puntuación calcula la suma de las puntuaciones locales del elemento de datos en todas las listas. El análisis de datos (TA) examina primero los elementos de datos que ocupan la posición 1 en todas las listas (d₁, d₂ y d₃). Busca las puntuaciones locales de estos elementos de datos en otras listas mediante acceso aleatorio y calcula sus puntuaciones generales (65, 63 y 70, respectivamente). Sin embargo, ninguno de ellos alcanza el umbral de la posición 1 (88).

Por lo tanto, en la posición 1, el TA no se detiene. En esta posición, tenemos Y = {d₁, d₂, d₃}, es decir, los k elementos de datos con la puntuación más alta observados hasta el momento. En las posiciones 2 y 3, Y se establece en {d₃, d₄, d₅} y {d₃, d₅, d₈}, respectivamente. Antes de la posición 6, ninguno de los elementos de datos involucrados en Y tiene una puntuación general mayor o igual al valor umbral. En la posición 6, el valor umbral es 63, que es menor que la puntuación general de los tres elementos de datos involucrados en Y, es decir, Y = {d₃, d₅, d₈}. Por lo tanto, el TA se detiene.

Nótese que el contenido de Y en la posición 6 es exactamente el mismo que en la posición 3. En otras palabras,

En la posición 3, Y ya contiene todas las k respuestas principales. En este ejemplo, TA realiza tres accesos ordenados adicionales en cada lista que no contribuyen al resultado final. Esto se caracteriza por una condición de parada conservadora que provoca que se detenga más tarde de lo necesario. En este ejemplo, realiza 9 accesos ordenados y $18 = (9 - 2)$ accesos aleatorios que no contribuyen al resultado final.

Algoritmos de estilo TA

Se han propuesto varios algoritmos de estilo TA, es decir, extensiones del algoritmo TATHreshold, para el procesamiento distribuido de consultas top-k. Los ilustramos mediante el algoritmo de Umbral Uniforme de Tres Fases (TPUT), que ejecuta consultas top-k en tres ciclos, suponiendo que cada lista está contenida en un nodo (al que llamamos el contenedor de la lista) y que la función de puntuación es suma. El algoritmo TPUT, ejecutado por el originador de la consulta, se detalla en el [Algoritmo 9.2](#).

TPUT funciona de la siguiente manera:

1. El creador de la consulta obtiene primero de cada contenedor de lista sus k elementos de datos principales. Sea f la función de puntuación, d un elemento de datos recibido y si(d) la puntuación local de d en la lista Li. La suma parcial de d se define como psum(d) = s i(d), donde s i(d) = si(d) ^{recibido} si el contenedor de Li ha enviado d al coordinador ; de lo contrario, s i(d) = 0. El creador de la consulta calcula las sumas parciales de todos los elementos de datos recibidos e identifica los elementos con las k sumas parciales más altas. La suma parcial del k-ésimo elemento de datos (denominado final de la fase 1) se denota por λ_1 .
2. El creador de la consulta envía un valor umbral $\tau = \lambda_1/m$ a cada titular de la lista. En respuesta, cada titular de la lista devuelve todos sus elementos de datos cuyas puntuaciones locales no sean inferiores a τ . La intuición es que si ningún nodo informa un elemento de datos en esta fase, su puntuación debe ser menor que λ_1 , por lo que no puede ser uno de los k elementos de datos principales. Sea Y el conjunto de datos recibidos de los titulares de la lista. El creador de la consulta calcula las nuevas sumas parciales de los datos en Y e identifica los elementos con las k sumas parciales más altas. La suma parcial del k-ésimo elemento (denominado punto final de la fase 2) se denota por λ^2 . Sea la puntuación del límite superior de un elemento de datos d definida como u(d) = ui(d), donde ui(d) = si(d) si se ha recibido d; de lo contrario, ui(d) = 0.
- Para cada elemento de datos d $\in D$, si u(d) es menor que λ_2 , se elimina de Y
 - Los datos restantes en Y se denominan candidatos top-k, ya que puede haber algunos datos en Y que no se hayan obtenido de todos los titulares de la lista. Es necesaria una tercera fase para recuperarlos.
3. El creador de la consulta envía el conjunto de los k elementos de datos candidatos principales a cada titular de la lista que devuelve sus puntuaciones. Luego, calcula la puntuación general, extrae los k elementos de datos con las puntuaciones más altas y devuelve la respuesta al usuario.

Ejemplo 9.4 Considere las dos primeras listas ordenadas (Lista 1 y Lista 2) en la [Figura 9.10](#).

Supongamos una consulta Q de las 2 principales, es decir, $k = 2$, donde la función de puntuación es la suma. Fase 1

Algoritmo 9.2: Umbral uniforme trifásico (TPUT)

Entrada: L₁, L₂, ..., L_m: m listas ordenadas de n elementos de datos, cada una en un contenedor de lista diferente f: función de puntuación Salida: Y : lista de los k elementos de datos principales

```

comienza {Fase 1} para i de 1 a m en
    |     paralelo Y ← recibe los k elementos de datos principales
    del
    contenedor Li fin para Z ← elementos de datos con la k suma
    parcial más alta en Y λ1 ← suma parcial del k-
        ésmo
    elemento de datos en Z {Fase 2} para i
        |     de 1 a m en paralelo envía λ1/m
        |         al contenedor de Li Y ← todos los elementos de datos del contenedor de Li cuyas puntuaciones
        locales
        no sean menores que λ1/m fin para Z ← elementos de datos con
        la k suma parcial más alta en Y λ2 ← suma parcial
        del k-ésimo elemento de datos en ZY ← Y - {elementos de datos en Y cuya puntuación de límite superior es menor que λ2}
        {Fase 3}
        para i de 1 a m en paralelo envía Y al
            |     titular de Li Z ← elementos
            |         de datos del titular de Li que están tanto en Y como en Li , y para Y ← k
            elementos
            de datos con la puntuación general más alta en el extremo Z

```

Produce los conjuntos Y = {d₁, d₂, d₄, d₆} y Z = {d₁, d₂}. El k-ésimo dato (es decir, el segundo) es d₂, cuya suma parcial es 28. Por lo tanto, obtenemos $\lambda_1/2 = 28/2 = 14$. Denotemos ahora cada dato d en Y como (d, puntuación en la Lista 1, puntuación en la Lista 2). La fase 2 produce Y = {(d₁, 30, 21), (d₂, 0, 28), (d₃, 26, 14), (d₄, 28, 0), (d₅, 17, 24), (d₆, 14, 27), (d₇, 25, 25), (d₈, 23, 20), (d₉, 27, 23)} y Z = {(d₁, 30, 21), (d₇, 25, 25)}. Cabe destacar que también se podría haber elegido d₉ en lugar de d₇ , ya que tiene la misma suma parcial.

Por lo tanto, obtenemos $\lambda_2/2 = 50$. Se obtienen las puntuaciones del límite superior de los elementos de datos en Y.

como:

$$\begin{aligned}
 u(d_1) &= 30 + 21 = 51 \quad u(d_2) = \\
 14 + 28 &= 42 \quad u(d_3) = 26 + 14 = \\
 40 \quad u(d_4) &= 28 + 14 = 42 \quad u(d_5) \\
 &= 17 + 24 = 41 \quad u(d_6) = 14 + 27 \\
 &= 41 \quad u(d_7) = 25 + 25 = 50 \quad u(d_8) \\
 &= 23 + 20 = 43 \quad u(d_9) = 27 + 23 \\
 &= 50
 \end{aligned}$$

Tras eliminar los elementos de datos en Y cuya puntuación límite superior es inferior a λ_2 , obtenemos $Y = \{d1, d7, d9\}$. La tercera fase no es necesaria en este caso, ya que todos los elementos de datos tienen todas sus puntuaciones locales. Por lo tanto, el resultado final es $Y = \{d1, d7\}$ o $Y = \{d1, d9\}$.

Cuando el número de listas (es decir, m) es alto, el tiempo de respuesta de TPUT es mucho mejor que el del algoritmo TA básico.

Algoritmo de mejor posición (BPA)

Existen muchas instancias de bases de datos en las que TA continúa explorando las listas a pesar de haber visto todas las respuestas top-k (como en el Ejemplo 9.3). Por lo tanto, es posible detenerse mucho antes. Basándose en esta observación, se han propuesto algoritmos de mejor posición (BPA) que ejecutan consultas top-k con mucha mayor eficiencia que TA. La idea clave de BPA es que el mecanismo de detención considera posiciones especiales en las listas, denominadas mejores posiciones. Intuitivamente, la mejor posición en una lista es la posición más alta, de modo que cualquier posición anterior también se haya visto. La condición de detención se basa en la puntuación general calculada utilizando las mejores posiciones en todas las listas.

La versión básica de BPA (véase el algoritmo 9.3) funciona como TA, excepto que registra todas las posiciones que se ven bajo acceso ordenado o aleatorio, calcula las mejores posiciones y tiene una condición de parada diferente. Para cada lista L_i , sea P_i el conjunto de posiciones que se ven bajo acceso ordenado o aleatorio en L_i . Sea bpi , la mejor posición en L_i , la posición más alta en P_i tal que cualquier posición de L_i entre 1 y bpi también esté en P_i . En otras palabras, bpi es mejor porque estamos seguros de que todas las posiciones de L_i entre 1 y bpi se han visto bajo acceso ordenado o aleatorio. Sea $s_i(bpi)$ la puntuación local del elemento de datos que está en la posición bpi en la lista L_i . Entonces, el umbral de BPA es $f(s_1(bp1), s_2(bp2), \dots, s_m(bpm))$ para alguna función f .

Ejemplo 9.5 Para ilustrar el BPA básico, considere nuevamente las tres listas ordenadas que se muestran en la Figura 9.10 y la consulta Q en el Ejemplo 9.3.

En la posición 1 , BPA visualiza los elementos de datos $d1, d2$ y $d3$. Para cada elemento visualizado, realiza un acceso aleatorio y obtiene su puntuación local y posición en todas las listas.

Por lo tanto, en este paso, las posiciones que se observan en la lista L_1 son las posiciones 1, 4 y 9, que son, respectivamente, las posiciones de $d1, d3$ y $d2$. Por lo tanto, tenemos $P_1 = \{1, 4, 9\}$ y la mejor posición en L_1 es $bp_1 = 1$ (ya que la siguiente posición es 4, lo que significa que las posiciones 2 y 3 no se han visto). Para L_2 y L_3 tenemos $P_2 = \{1, 6, 8\}$ y $P_3 = \{1, 5, 8\}$, por lo que $bp_2 = 1$ y $bp_3 = 1$. Por lo tanto, la puntuación general de las mejores posiciones es $\lambda = f(s_1(1), s_2(1), s_3(1)) = 30 + 28 + 30 = 88$. En la posición 1, el conjunto de los tres elementos de datos con la puntuación más alta es $Y = \{d1, d2, d3\}$, y dado que la puntuación general de estos elementos de datos es menor que λ

, El BPA no puede detenerse.

2. En la posición 2, BPA ve $d4, d5$ y $d6$. Por lo tanto, tenemos $P_1 = \{1, 2, 4, 7, 8, 9\}, P_2 = \{1, 2, 4, 6, 8, 9\}$ y $P_3 = \{1, 2, 4, 5, 6, 8\}$. Por lo tanto, tenemos $bp_1 = 2, bp_2 = 2$ y $bp_3 = 2$, por lo que $\lambda = f(s_1(2), s_2(2), s_3(2)) = 28 + 27 + 29 = 84$.

La puntuación general de los elementos de datos involucrados en $Y = \{d3, d4, d5\}$ es menor que 84, por lo que BPA no se detiene.

Algoritmo 9.3: Algoritmo de mejor posición (BPA)

Entrada: L1, L2,...,Lm: m listas ordenadas de n elementos de datos

f : función de puntuación

Salida: Y : lista de los k elementos de datos

```

principales
    comienza j ← 1
    umbral ← 1 min_overall_score
    ← 0 para i de 1 a m en paralelo hacer
        |   Pi ←
    fin para

    mientras j = n + 1 y min_overall_score < umbral hacer
        {Realice acceso ordenado en paralelo a cada una de las m listas
        ordenadas} para i desde 1 hasta m en paralelo
            |   {Procesar cada elemento de datos en la
            |   posición j} para cada elemento de datos d en la
            |   posición j en Li hacer {acceder a los puntajes locales de d en las otras listas a través
            |   de acceso aleatorio} puntuación_general(d) ← f (puntajes de d en cada Li)
        fin para

        Pi ← Pi   {posiciones vistas bajo acceso ordenado o aleatorio} bpi ←
        mejor posición en el extremo Li
    para

    Y ← k elementos de datos con la puntuación más alta
    hasta el momento min_overall_score ← puntuación general más pequeña de
    elementos de datos en Y umbral ← f (puntuaciones locales en la posición
    bpi en cada
    Li) j ← j + 1 fin mientras
fin

```

3. En la posición 3, BPA ve d7, d8 y d9. Por lo tanto, tenemos $P1 = P2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y $P3 = \{1, 2, 3, 4, 5, 6, 7, 8, 10\}$. Por lo tanto, tenemos $bp1 = 9$, $bp2 = 9$ y $bp3 = 8$. La puntuación total de las mejores posiciones es $\lambda = f(s1(9), s2(9), s3(8)) = 11+13+14 = 38$. En esta posición, tenemos $Y = \{d3, d5, d8\}$. Dado que la puntuación de todos los datos involucrados en Y es mayor que λ , BPA se detiene, es decir, exactamente en la primera posición donde BPA tiene todas las k respuestas principales.

Recordemos que sobre esta base de datos, TA se detiene en la posición 6.

Se ha demostrado que, para cualquier conjunto de listas ordenadas, BPA se detiene tan pronto como TA, y su coste de ejecución nunca es superior a TA. También se ha demostrado que el coste de ejecución de BPA puede ser $(m - 1)$ veces (donde m es el número de listas ordenadas) menor que el de TA. Aunque BPA es bastante eficiente, sigue realizando trabajo redundante. Una de las redundancias de BPA (y también de TA) es que puede acceder a algunos elementos de datos varias veces bajo acceso ordenado en diferentes listas. Por ejemplo, un elemento de datos al que se accede en una posición de una lista mediante acceso ordenado y, por lo tanto, al que se accede en otras listas mediante acceso aleatorio, puede volver a accederse en las otras listas mediante acceso ordenado en las siguientes posiciones. Un algoritmo mejorado, BPA2, evita esto y, por lo tanto, es mucho más eficiente que BPA. No transfiere las posiciones vistas de los propietarios de las listas a

El originador de la consulta. Por lo tanto, no necesita mantener las posiciones vistas ni sus puntuaciones locales. Además, accede a cada posición de una lista como máximo una vez.

El número de accesos a las listas realizados por BPA2 puede ser aproximadamente $(m-1)$ veces menor que el de BPA.

9.3.1.2 Consultas Top-k en sistemas no estructurados

Un posible enfoque para procesar consultas top-k en sistemas no estructurados consiste en enrutar la consulta a todos los pares, recuperar todas las respuestas disponibles, puntuarlas mediante la función de puntuación y devolver al usuario las k respuestas con la puntuación más alta. Sin embargo, este enfoque no es eficiente en términos de tiempo de respuesta ni de coste de comunicación.

La primera solución eficiente propuesta es PlanetP, un sistema P2P no estructurado. En PlanetP, un servicio de publicación/suscripción direccional por contenido replica datos en comunidades P2P de hasta diez mil pares. El algoritmo de procesamiento de consultas top-k funciona de la siguiente manera: dada una consulta Q, el emisor calcula una clasificación de relevancia de los pares con respecto a Q, los contacta uno por uno en orden descendente y les solicita que devuelvan un conjunto de sus datos con mayor puntuación, junto con sus puntuaciones. Para calcular la relevancia de los pares, se utiliza un índice global completamente replicado que contiene asignaciones de término a par. Este algoritmo tiene un rendimiento excelente en sistemas de escala moderada. Sin embargo, en un sistema P2P de gran tamaño, mantener el índice replicado actualizado puede afectar la escalabilidad.

Describimos otra solución desarrollada en el contexto de APPA, un sistema de gestión de datos independiente de la red P2P. Se ha propuesto un marco totalmente distribuido para ejecutar consultas top-k que también aborda la volatilidad de los pares durante la ejecución de consultas y aborda situaciones en las que algunos pares abandonan el sistema antes de finalizar el procesamiento de la consulta. Dada una consulta top-K Q con un TTL específico, el algoritmo básico denominado Top-k Totalmente Descentralizado (FD) funciona de la siguiente manera (véase el [Algoritmo 9.4](#)):

1. Reenvío de consultas. El originador de la consulta reenvía Q a los pares accesibles cuyos La distancia de salto desde el originador de la consulta es menor que TTL.
2. Ejecución y espera de la consulta local. Cada par p que recibe Q la ejecuta localmente: accede a los datos locales que coinciden con el predicado de la consulta, los califica mediante una función de puntuación, selecciona los k datos principales y los guarda localmente, junto con sus puntuaciones. A continuación, p espera recibir los resultados de sus vecinos. Sin embargo, dado que algunos vecinos pueden abandonar el sistema P2P y nunca enviar una lista de puntuaciones a p, el tiempo de espera tiene un límite que se calcula para cada par en función del TTL recibido, los parámetros de red y los parámetros de procesamiento local del par.
3. Fusión y retroalimentación. En esta fase, las puntuaciones más altas se transfieren al originador de la consulta mediante un algoritmo basado en trie, como se indica a continuación. Una vez transcurrido el tiempo de espera, p fusiona sus k puntuaciones más altas locales con las recibidas de sus vecinos y envía el resultado a su padre (el par del que recibió Q) en forma de lista de puntuaciones. Para minimizar el tráfico de red, FD no transfiere los datos principales.

Algoritmo 9.4: Top-k totalmente descentralizado (FD)

Entrada: Q: consulta top-k f:
 función de puntuación
 TTL: tiempo de vida w:
 tiempo de espera

Salida: Y: lista de los k elementos de datos principales que comienzan

```

En el origen de la consulta, el
par
    comienza a enviar Q a los vecinos
        F_inal_score_list ← fusionar listas de puntuaciones locales recibidas de los vecinos para cada
        par p en F_inal_score_list
            |   Y ← recupera los k elementos de datos principales
            en p final para
        fin
    para cada par que recibe Q de un par p
        TTL ← TTL - 1 si TTL >
        O entonces envía Q a
            |   los vecinos y termina si
        Local_score_list ← extrae las mejores puntuaciones locales
        Espera un tiempo
        Lista_de_puntuaciones_locales ← Lista_de_puntuaciones_locales   puntuaciones recibidas top-k
        Enviar Local_score_list al final p para
fin

```

Elementos (que podrían ser grandes), solo sus puntuaciones y direcciones. Una lista de puntuaciones es simplemente una lista de k pares (a, s), donde a es la dirección del par propietario del elemento de datos y s su puntuación.

4. Recuperación de datos. Tras recibir las listas de puntuaciones de sus vecinos, el originador de la consulta crea la lista de puntuaciones final fusionando sus k puntuaciones máximas locales con las listas de puntuaciones fusionadas recibidas de sus vecinos. A continuación, recupera directamente los k elementos de datos principales de los pares que los contienen.

El algoritmo es completamente distribuido y no depende de la existencia de pares específicos, lo que permite abordar la volatilidad de los pares durante la ejecución de consultas. En particular, se abordan los siguientes problemas: la inaccesibilidad de los pares en la fase de fusión y reversión; la inaccesibilidad de los pares que contienen los elementos de datos principales en la fase de recuperación de datos; y la recepción tardía de las listas de puntuación por parte de un par una vez transcurrido su tiempo de espera. La evaluación del rendimiento de FD muestra que puede lograr importantes mejoras en términos de coste de comunicación y tiempo de respuesta.

9.3.1.3 Consultas Top-k en DHT

Como ya comentamos, la principal función de un DHT es asignar un conjunto de claves a los pares del sistema P2P y buscar eficientemente al par responsable de una clave dada. Esto ofrece un soporte eficiente y escalable para consultas de coincidencia exacta.

Sin embargo, soportar consultas top-k sobre DHT no es sencillo. Una solución sencilla consiste en recuperar todas las tuplas de las relaciones involucradas en la consulta, calcular la puntuación de cada tupla recuperada y, finalmente, devolver las k tuplas con las puntuaciones más altas. Sin embargo, esta solución no puede escalarse a un gran número de tuplas almacenadas. Otra solución consiste en almacenar todas las tuplas de cada relación con la misma clave (p. ej., el nombre de la relación), de modo que todas las tuplas se almacenen en el mismo par. De este modo, el procesamiento de consultas top-k puede realizarse en ese par central mediante algoritmos centralizados conocidos. Sin embargo, el par se convierte en un cuello de botella y un punto único de fallo.

Como parte del proyecto APPA, se ha propuesto una solución basada en TA (véase la sección [9.3.1.1](#)) y un mecanismo que almacena los datos compartidos en la DHT de forma totalmente distribuida. En APPA, los pares pueden almacenar sus tuplas en la DHT mediante dos métodos complementarios: almacenamiento de tuplas y almacenamiento de valores de atributos. Con el almacenamiento de tuplas, cada tupla se almacena en la DHT utilizando su identificador (p. ej., su clave principal) como clave de almacenamiento. Esto permite buscar una tupla por su identificador, de forma similar a un índice principal. El almacenamiento de valores de atributos almacena individualmente en la DHT los atributos que pueden aparecer en el predicado de igualdad o en la función de puntuación. Por lo tanto, al igual que en los índices secundarios, permite buscar las tuplas utilizando sus valores de atributo. El almacenamiento de valores de atributo tiene dos propiedades importantes: (1) después de recuperar un valor de atributo del DHT, los pares pueden recuperar fácilmente la tupla correspondiente del valor del atributo; (2) los valores de atributo que son relativamente "cercanos" se almacenan en el mismo par. Para proporcionar la primera propiedad, la clave, que se utiliza para almacenar la tupla completa, se almacena junto con el valor del atributo. La segunda propiedad se proporciona utilizando el concepto de partición de dominio de la siguiente manera. Considere un atributo a y sea Da su dominio de valores. Suponga que hay un orden total \leq en Da (p. ej., Da es numérico). Da se partitiona en n subdominios no vacíos d_1, d_2, \dots, d_n tales que su unión es igual a Da , la intersección de cualesquier dos subdominios diferentes está vacía y para cada $v_1 \in d_i$ y $v_2 \in d_j$, si $i < j$, entonces tenemos $v_1 \leq v_2$. La función hash se aplica al subdominio del valor del atributo. Por lo tanto, para los valores de atributo que pertenecen al mismo subdominio, la clave de almacenamiento es la misma y se almacenan en el mismo par. Para evitar la distorsión en el almacenamiento de atributos (es decir, la distribución desigual de los valores de atributo dentro de los subdominios), la partición del dominio se realiza de forma que los valores de atributo se distribuyan uniformemente en los subdominios. Esta técnica utiliza información basada en histogramas que describe la distribución de los valores del atributo.

Usando este modelo de almacenamiento, el algoritmo de procesamiento de consultas top-k, llamado DHTop (ver Algoritmo 9.5), funciona de la siguiente manera. Sea Q una consulta top-k dada, f su función de puntuación y p_0 el par en el que se emite Q . Para simplificar, supongamos que f es una función de puntuación monótona. Sea los atributos de puntuación el conjunto de atributos que se pasan a la función de puntuación como argumentos. DHTop comienza en p_0 y procede en dos fases: primero prepara listas ordenadas de subdominios candidatos y luego...

Algoritmo 9.5: DHT Top-k (DHTop)

```

Entrada: Q: consulta top-k; f:
función de puntuación;
A: conjunto de m atributos utilizados en f
Salida: Y: lista de las k tuplas superiores
comienzan
    {Fase 1: preparar listas de subdominios de atributos} para cada
    atributo de puntuación Ai en A
        LAi ← todos los subdominios de Ai
        LAi ← LAi– subdominios que no satisfacen la condición de Q
        Ordenar LAi en orden descendente de sus subdominios al final

    {Fase 2: recuperar continuamente los valores de los atributos y sus tuplas hasta encontrar las k tuplas
    superiores}
    Hecho ← falso
    para cada atributo de puntuación Ai en A en paralelo
        yo ← 1
        mientras (i < número de subdominios de A) y no Done envía Q al par p
            que mantiene los valores de los atributos del subdominio i en LAi Z ← Ai valores (en orden
            descendente) de p que satisfacen la condición de Q, junto con sus claves de almacenamiento
            de datos correspondientes para cada valor recibido v obtiene
            la tupla de v Y ← k tuplas con la
                puntuación más alta
                hasta el momento umbral ← f (v1, v2,...,vm) tal
                que vi es el último valor recibido para el atributo Ai en A min_overall_score ← puntuación
                general más pequeña de
                las tuplas en Y si min_overall_score ≤ umbral entonces Done ← verdadero
                fin si i ← i + 1 fin para
                |
        terminar mientras
    fin para
fin

```

Recupera continuamente los valores de los atributos candidatos y sus tuplas hasta encontrar las k tuplas principales. Los detalles de los dos pasos son los siguientes:

1. Para cada atributo de puntuación A_i , p_0 prepara la lista de subdominios y los ordena en orden descendente según su impacto positivo en la función de puntuación. Para cada lista, p_0 elimina los subdominios en los que ningún miembro cumple las condiciones de Q . Por ejemplo, si existe una condición que exige que el atributo de puntuación sea igual a una constante (p. ej., $A_i = 10$), p_0 elimina todos los subdominios excepto el que corresponde a la constante. Denotemos con LAi la lista preparada en esta fase para el atributo de puntuación A_i .
2. Para cada atributo de puntuación A_i , en paralelo, p_0 procede de la siguiente manera. Envía Q y A_i al par, digamos p , responsable de almacenar los valores del primer subdominio de LAi , y le solicita que devuelva los valores de A_i en p . Los valores son

Se devuelven a p0 según su impacto positivo en la función de puntuación. Tras recibir cada valor de atributo, p0 recupera su tupla correspondiente, calcula su puntuación y la conserva si se encuentra entre las k más altas calculadas hasta la fecha. Este proceso continúa hasta obtener k tuplas cuyas puntuaciones superen un umbral calculado con base en los valores de atributo recuperados hasta el momento. Si los valores de atributo que p devuelve a p0 no son suficientes para determinar las k tuplas principales, p0 envía Q y Ai al sitio responsable del segundo subdominio de LAi , y así sucesivamente hasta encontrar las k tuplas principales.

Sean A1, A2,..., Am los atributos de puntuación y v1, v2,...,vm los últimos valores recuperados, respectivamente, para cada uno de ellos. El umbral se define como $\tau = f(v_1, v_2, \dots, v_m)$. Una característica principal de DHTop es que, tras recuperar cada nuevo valor de atributo, el valor del umbral disminuye. Por lo tanto, tras recuperar un número determinado de valores de atributo y sus tuplas, el umbral se vuelve menor que k elementos de datos recuperados y el algoritmo se detiene. Se ha demostrado analíticamente que DHTop funciona correctamente tanto para funciones de puntuación monótonas como para un amplio grupo de funciones no monótonas.

9.3.1.4 Consultas Top-k en sistemas Superpeer

Un algoritmo típico para el procesamiento de consultas top-k en sistemas superpeer es el de Edutella. En Edutella, un pequeño porcentaje de nodos son superpeers y se asume que tienen alta disponibilidad y una excelente capacidad de cómputo. Los superpeers son responsables del procesamiento de consultas top-k, mientras que los demás pares solo ejecutan las consultas localmente y puntúan sus recursos. El algoritmo es bastante simple y funciona de la siguiente manera: Dada una consulta Q, el originador de la consulta envía Q a su superpeer, que a su vez la envía a los demás superpeers. Los superpeers reenvían Q a los pares relevantes conectados a ellos. Cada par que tenga elementos de datos relevantes para Q los puntúa y envía el elemento de datos con la puntuación más alta a su superpeer. Cada superpeer elige el elemento con la puntuación más alta de todos los elementos de datos recibidos. Para determinar el segundo mejor elemento, solo le pide a un par, el que ha devuelto el primer elemento con la puntuación más alta, que devuelva su segundo elemento con la puntuación más alta. El superpeer selecciona el segundo elemento con la puntuación más alta de los elementos recibidos previamente y el elemento recién recibido. Luego, pregunta al par que devolvió el segundo elemento superior, y así sucesivamente hasta recuperar los k elementos superiores. Finalmente, los superpeers envían sus elementos superiores al superpeer del originador de la consulta para extraer los k elementos superiores totales y enviárselos al originador de la consulta.

Este algoritmo minimiza la comunicación entre pares y superpar ya que, después de haber recibido los elementos de datos con puntuación máxima de cada par conectado a él, cada superpar solicita solo a un par el siguiente elemento superior.

9.3.2 Consultas de unión

Los algoritmos de unión más eficientes en bases de datos distribuidas y paralelas se basan en hash. Por lo tanto, el hecho de que una DHT dependa del hash para almacenar y localizar datos puede aprovecharse de forma natural para admitir consultas de unión eficientemente. Se ha propuesto una solución básica en el contexto del sistema P2P PIER que admite consultas complejas sobre DHT. La solución es una variación del algoritmo de unión hash paralelo (PHJ) (véase la sección 8.4.1), que denominamos PIERjoin. Al igual que en el algoritmo PHJ, PIERjoin asume que las relaciones unidas y las relaciones resultantes tienen un nodo principal (denominado espacio de nombres en PIER), que son los nodos que almacenan fragmentos horizontales de la relación. A continuación, utiliza el método put para distribuir tuplas en un conjunto de pares en función de su atributo de unión, de modo que las tuplas con los mismos valores de atributo de unión se almacenen en los mismos pares. Para realizar uniones localmente, PIER implementa una versión del algoritmo de unión hash simétrica (véase la sección 8.4.1.2) que proporciona un soporte eficiente para el paralelismo segmentado. En la unión hash simétrica, con dos relaciones de unión, cada nodo que recibe tuplas para unir mantiene dos tablas hash, una por relación.

Por lo tanto, al recibir una nueva tupla de cualquiera de las relaciones, el nodo la añade a la tabla hash correspondiente y la compara con la tabla hash opuesta basándose en las tuplas recibidas hasta el momento. PIER también depende de la DHT para gestionar el comportamiento dinámico de los pares (entradas o salidas de la red durante la ejecución de la consulta) y, por lo tanto, no garantiza la integridad de los resultados.

Para una consulta de unión binaria Q (que puede incluir predicados de selección), PIERjoin funciona en tres fases (ver Algoritmo 9.6): multidifusión, hash y sondeo/unión.

1. Fase de multidifusión. El par originador de la consulta multidifunde Q a todos los pares que almacenan tuplas de las relaciones de unión R y S, es decir, sus ubicaciones.
2. Fase de hash. Cada par que recibe Q escanea su relación local, buscando las tuplas que satisfacen el predicado de selección (si las hay). Luego, envía las tuplas seleccionadas al inicio de la relación resultante mediante operaciones de colocación . La clave DHT utilizada en la operación de colocación se calcula utilizando el inicio de la relación resultante y el atributo de unión.
3. Fase de sondeo/unión. Cada par en el origen de la relación resultante, al recibir una nueva tupla, la inserta en la tabla hash correspondiente, sondea la tabla hash opuesta para encontrar tuplas que coincidan con el predicado de unión (y un predicado de selección, si lo hay), y construye las tuplas resultantes unidas. Recordemos que el "origen" de una relación (particionada horizontalmente) se definió en el capítulo 4 como un conjunto de pares donde cada uno tiene una partición diferente. En este caso, la partición se realiza mediante hash en el atributo de unión. El origen de la relación resultante también es una relación particionada (mediante operaciones de colocación), por lo que también se encuentra en múltiples pares.

Este algoritmo básico se puede mejorar de varias maneras. Por ejemplo, si una de las relaciones ya tiene un hash en los atributos de unión, podemos usar su origen como origen del resultado, utilizando una variante del algoritmo de unión asociativa paralela (PAJ) (véase la sección 8.4.1), donde solo es necesario aplicar un hash a una relación y enviarla a través del DHT.

Algoritmo 9.6: PIERjoin Entrada: Q:

consulta de unión sobre las relaciones R y S en el atributo A; h: función hash; HR, HS:
 hogares de R y S Salida: T:
 relación de resultado de unión; HT:
 hogar de T begin

{Fase de multidifusión}

En el par de origen de la consulta, envía Q a todos los pares en HR y HS {fase hash}

para cada par p en HR que recibió Q en paralelo .

- para cada tupla r en Rp que satisface el predicado de selección, coloque r usando $h(HT, A)$ y así sucesivamente.

fin para

cada par p en HS que recibió Q en paralelo

- para cada tupla s en Sp que satisface el predicado de selección, coloque s usando $h(HT, A)$ y así sucesivamente.

fin para

{Fase de sondeo/unión}

para cada par p en HT en paralelo si ha llegado una nueva tupla i , entonces si i es una tupla r , entonces sondea las tuplas s en Sp usando $h(A)$ de lo contrario sondear r tuplas en Rp usando $h(A)$

fin si

$T_p \leftarrow rs$ termina

si fin para

fin

9.3.3 Consultas de rango

Recuerde que las consultas de rango tienen una cláusula WHERE con la forma "atributo A en el rango $[a, b]$ ", donde a y b son valores numéricos. Los sistemas P2P estructurados, en particular las DHT, son muy eficientes al admitir consultas de coincidencia exacta (de la forma " $A = a$ "), pero presentan dificultades con las consultas de rango. La razón principal es que el hash tiende a alterar el orden de los datos, útil para encontrar rangos rápidamente.

Existen dos enfoques principales para admitir consultas de rango en sistemas P2P estructurados: extender un DHT con propiedades de proximidad o preservación del orden, o mantener el orden de claves con una estructura basada en trie. El primer enfoque se ha utilizado en varios sistemas. El hash sensible a la localidad es una extensión de los DHT que aplica hashes a rangos similares en el mismo nodo DHT con alta probabilidad. Sin embargo, este método solo puede obtener respuestas aproximadas y puede causar cargas desequilibradas en redes grandes.

El Árbol Hash de Prefijos (PHT) es una estructura de datos distribuida basada en trie que admite consultas de rango sobre un DHT, simplemente usando la operación de búsqueda DHT. Los datos que se indexan son cadenas binarias de longitud D. Cada nodo tiene 0 o 2 hijos, y una clave k se almacena en un nodo hoja cuya etiqueta es un prefijo de k. Además, los nodos hoja están vinculados a sus vecinos. La operación de búsqueda de PHT en la clave k debe devolver el nodo hoja único hoja (k) cuya etiqueta es un prefijo de k. Dada una clave k de longitud D, hay D + 1 prefijos distintos de k. La obtención de hoja (k) puede realizarse mediante un escaneo lineal de estos posibles D + 1 nodos. Sin embargo, dado que un PHT es un trie binario, el escaneo lineal puede mejorarse mediante una búsqueda binaria en la longitud del prefijo. Esto reduce el número de búsquedas DHT de (D + 1) a $\log D$. Dadas dos claves a y b, como $a \leq b$, se admiten dos algoritmos para consultas de rango mediante la búsqueda de PHT. El primero es secuencial: busca la hoja (a) y luego escanea secuencialmente la lista enlazada de nodos hoja hasta llegar al nodo hoja (b). El segundo algoritmo es paralelo: primero identifica el nodo correspondiente al rango de prefijo más pequeño que cubre completamente el rango [a, b]. Para llegar a este nodo, se utiliza una búsqueda DHT simple y la consulta se reenvía recursivamente a los hijos que se superponen con el rango [a, b].

Como en todos los esquemas de hash, el primer enfoque presenta un sesgo de datos que puede generar pares con rangos desequilibrados, lo que perjudica el equilibrio de carga. Para solucionar este problema, el segundo enfoque aprovecha las estructuras basadas en trie para mantener rangos de claves equilibrados. El primer intento de construir una red P2P basada en una estructura trie equilibrada es BATON (Red de Superposición de Árbol Equilibrado). A continuación, presentamos BATON y su compatibilidad con consultas de rango con más detalle.

BATON organiza los pares como un trie binario balanceado (cada nodo del trie es mantenido por un par). La posición de un nodo en BATON está determinada por una tupla (nivel, número), con el nivel comenzando desde 0 en la raíz, el número comenzando desde 1 en la raíz y asignado secuencialmente mediante recorrido en orden. Cada nodo del trie almacena enlaces a su padre, hijos, nodos adyacentes y nodos vecinos seleccionados que son nodos del mismo nivel. Dos tablas de enrutamiento: una tabla de enrutamiento izquierda y una tabla de enrutamiento derecha almacenan enlaces a los nodos vecinos seleccionados. Para un nodo numerado i, estas tablas de enrutamiento contienen enlaces a nodos ubicados en el mismo nivel con números que son menores (tabla de enrutamiento izquierda) y mayores (tabla de enrutamiento derecha) que i por una potencia de 2. El elemento j en la tabla de enrutamiento izquierda (derecha) en el nodo i contiene un enlace al nodo numerado $i - 2^j - 1$ ($i + 2^j - 1$) en el mismo nivel en el trie. La figura 9.11 muestra la tabla de enrutamiento del nodo 6.

En BATON, a cada hoja y nodo interno (o par) se le asigna un rango de valores. Para cada enlace, este rango se almacena en la tabla de enrutamiento y, cuando cambia, el enlace se modifica para registrar el cambio. El rango de valores gestionado por un par debe estar a la derecha del rango gestionado por su subárbol izquierdo y ser menor que el rango gestionado por su subárbol derecho (véase la figura 9.12). Por lo tanto, BATON crea una estructura de índice distribuido eficaz. La entrada y salida de pares se procesa de forma que el trie se mantenga equilibrado, reenviando la solicitud hacia arriba en el trie para las entradas.

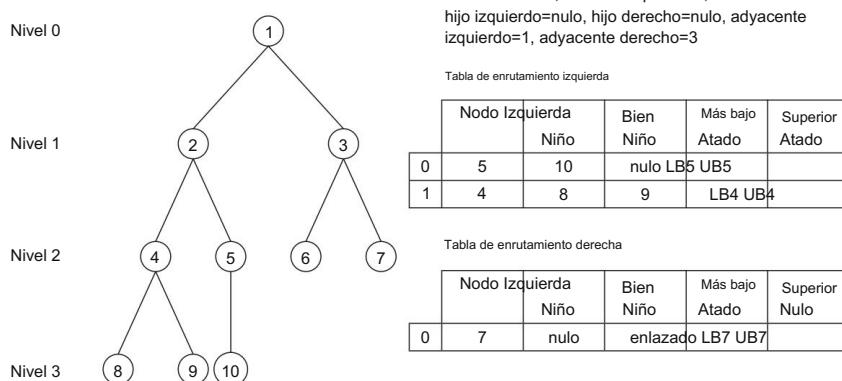


Fig. 9.11 Índice del árbol de estructura BATON y tabla de enruteamiento del nodo 6

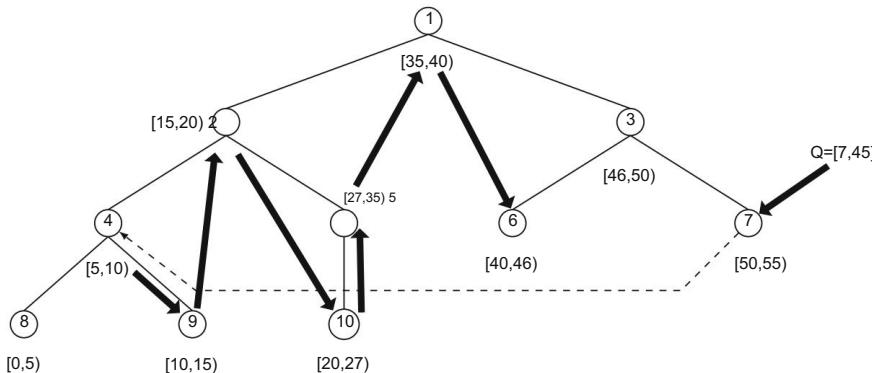


Fig. 9.12 Procesamiento de consultas de rango en BATON

y hacia abajo en el trie para las hojas, por lo tanto con no más de $O(\log n)$ pasos para un trie de n nodos.

Una consulta de rango se procesa de la siguiente manera (Algoritmo 9.7). Para una consulta de rango Q con rango $[a, b]$ enviada por el nodo i , se busca un nodo que interseca con el límite inferior del rango buscado. El par que almacena el límite inferior del rango busca localmente tuplas pertenecientes al rango y reenvía la consulta a su nodo adyacente derecho. En general, cada nodo que recibe la consulta busca tuplas locales y contacta con su nodo adyacente derecho hasta alcanzar el nodo que contiene el límite superior del rango. Las respuestas parciales obtenidas al encontrar una intersección se envían al nodo que envía la consulta. La primera intersección se encuentra en $O(\log n)$ pasos utilizando un algoritmo para consultas de coincidencia exacta. Por lo tanto, una consulta de rango con X nodos que cubren el rango se responde en $O(\log n + X)$ pasos.

Algoritmo 9.7: BatonRange Entrada: Q:

una consulta de rango en el formato $[a, b]$

Salida: T: resultado relación

inicio

{Buscar el par que almacena el límite inferior del rango}

En el originador de la consulta,

el par

comienza a buscar el par p que

tiene el valor

a,

envía Q al p y termina por cada par p que

recibe Q, haz $T_p \leftarrow Range(p) \cap [a, b]$, envía T_p al originador de la

consulta si $Range(RightAdj\ acent(p)) \cap [a, b] = \emptyset$, entonces

deja que p sea el par adyacente

derecho de p,

envía Q al p y termina si

fin para

fin

Ejemplo 9.6 Considere la consulta Q con rango $[7, 45]$ emitida en el nodo 7 en la Figura 9.12.

Primero, BATON ejecuta una consulta de coincidencia exacta buscando un nodo que contenga el límite inferior del rango (ver línea discontinua en la figura). Dado que el límite inferior se encuentra en el rango asignado al nodo 4, busca localmente tuplas pertenecientes al rango y reenvía la consulta a su nodo adyacente derecho (nodo 9). El nodo 9 busca tuplas locales pertenecientes al rango y reenvía la consulta al nodo 2. Los nodos 10, 5, 1 y 6 reciben la consulta, buscan tuplas locales y contactan con su respectivo nodo adyacente derecho hasta alcanzar el nodo que contiene el límite superior del rango.

9.4 Consistencia de la réplica

Para aumentar la disponibilidad de los datos y el rendimiento del acceso, los sistemas P2P replican los datos. Sin embargo, los diferentes sistemas P2P proporcionan niveles muy diferentes de consistencia de réplica. Los sistemas P2P más simples, como Gnutella y Kazaa, solo procesan datos estáticos (p. ej., archivos de música) y la replicación es pasiva, ya que ocurre de forma natural a medida que los pares solicitan y copian archivos entre sí (básicamente, almacenan datos en caché). En sistemas P2P más avanzados, donde las réplicas se pueden actualizar, se requieren técnicas adecuadas de gestión de réplicas. Desafortunadamente, la mayor parte del trabajo sobre la consistencia de las réplicas se ha realizado únicamente en el contexto de las DHT. Podemos distinguir tres enfoques para gestionar la consistencia de las réplicas: soporte básico en las DHT, actualización de datos en las DHT y conciliación de réplicas. En esta sección, presentamos las principales técnicas utilizadas en estos enfoques.

9.4.1 Soporte básico en DHT

Para mejorar la disponibilidad de datos, la mayoría de los DHT se basan en la replicación de datos mediante el almacenamiento de pares (clave, datos) en varios pares mediante, por ejemplo, el uso de varias funciones hash. Si un par no está disponible, sus datos aún pueden recuperarse de los demás pares que tienen una réplica. Algunas DHT proporcionan soporte básico para que la aplicación gestione la consistencia de la réplica. En esta sección, describimos las técnicas utilizadas en dos DHT populares: CAN y Tapestry.

CAN ofrece dos enfoques para la replicación. El primero consiste en usar m funciones hash para mapear una clave única en m puntos del espacio de coordenadas y, en consecuencia, replicar un único par (clave, datos) en m nodos distintos de la red. El segundo enfoque es una optimización del diseño básico de CAN. Consiste en que un nodo envía proactivamente claves populares a sus vecinos cuando detecta una sobrecarga de solicitudes. En este enfoque, las claves replicadas deben tener un campo TTL asociado para revertir automáticamente el efecto de la replicación al final del período de sobrecarga. Además, la técnica asume datos inmutables (de solo lectura).

Tapestry es un sistema P2P extensible que proporciona localización y enrutamiento de objetos descentralizados sobre una red superpuesta estructurada. Enruta mensajes a puntos finales lógicos (es decir, puntos finales cuyos identificadores no están asociados con la ubicación física), como nodos o réplicas de objetos. Esto permite la entrega de mensajes a puntos finales móviles o replicados en presencia de inestabilidad en la infraestructura subyacente. Además, Tapestry considera la latencia para establecer la vecindad de cada nodo. Los mecanismos de ubicación y enrutamiento de Tapestry funcionan de la siguiente manera. Sea o un objeto identificado por $\text{id}(o)$; la inserción de o en la red P2P involucra dos nodos: el nodo servidor (ns), que contiene o , y el nodo raíz (nr), que contiene una asignación con la forma $(\text{id}(o), ns)$, que indica que el objeto identificado por $\text{id}(o)$ se almacena en el nodo ns . El nodo raíz se determina dinámicamente mediante un algoritmo determinista globalmente consistente. La Figura 9.13a muestra que cuando o se inserta en ns , ns publica $\text{id}(o)$ en su nodo raíz enrutando un mensaje de ns a nr que contiene la asignación $(\text{id}(o), ns)$. Esta asignación se almacena en todos los nodos a lo largo de la ruta del mensaje. Durante una consulta de ubicación, por ejemplo, "¿ $\text{id}(o)$?" en la Fig. 9.13a, el mensaje que busca $\text{id}(o)$ se enruta inicialmente hacia nr , pero puede detenerse antes de alcanzarlo una vez que se encuentra un nodo que contiene la asignación $(\text{id}(o), ns)$. Para enrutar un mensaje a la raíz de $\text{id}(o)$, cada nodo reenvía este mensaje a su vecino cuyo identificador lógico sea el más similar a $\text{id}(o)$.

Tapestry ofrece toda la infraestructura necesaria para aprovechar las réplicas, como se muestra en la Fig. 9.13b. Cada nodo del grafo representa un par en la red P2P y contiene su identificador lógico en formato hexadecimal. En este ejemplo, dos réplicas, $O1$ y $O2$, del objeto O (p. ej., un archivo de libro) se insertan en pares distintos ($O1 \rightarrow$ par 4228 y $O2 \rightarrow$ par AA93). El identificador de $O1$ es igual al de $O2$ (es decir, 4378 en hexadecimal), ya que $O1$ y $O2$ son réplicas del mismo objeto O . Cuando $O1$ se inserta en su nodo servidor (par 4228), la asignación (4378, 4228) se enruta del par 4228 al par 4377 (el nodo raíz para

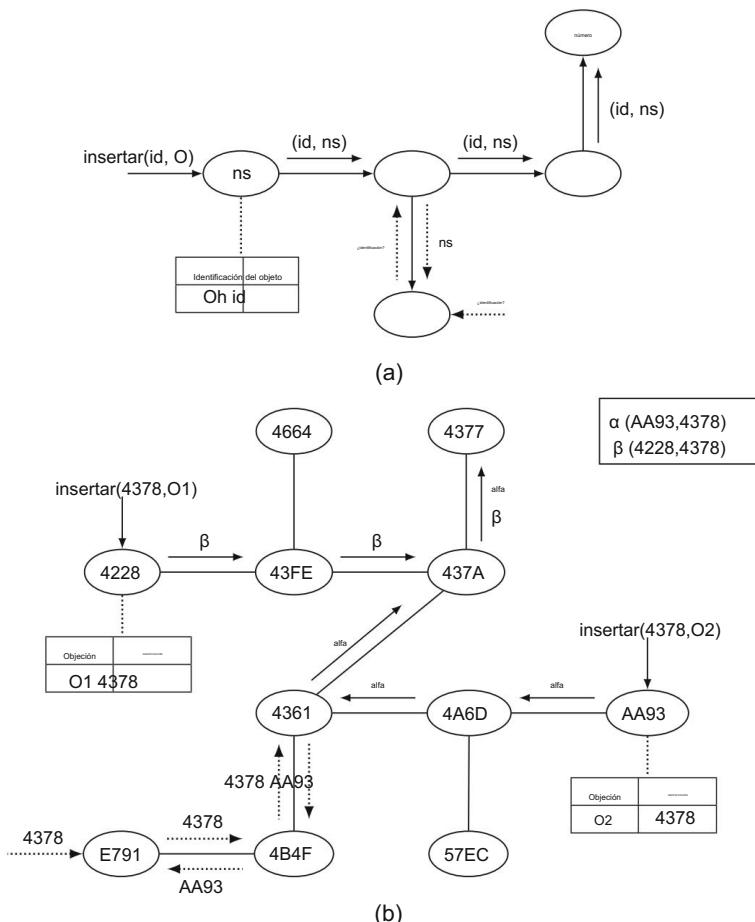


Fig. 9.13 Tapestry. (a) Publicación de objetos. (b) Gestión de réplicas

Identificador de O_1). A medida que el mensaje se acerca al nodo raíz, el objeto y el nodo Los identificadores se vuelven cada vez más similares. Además, la asignación $(4378, 4228)$ es almacenados en todos los pares a lo largo de la ruta del mensaje. La inserción de O_2 sigue el mismo proceso. procedimiento. En la figura 9.13b, si el par $E791$ busca una réplica de O , el asociado El enrutamiento de mensajes se detiene en el par 4361 . Por lo tanto, las aplicaciones pueden replicar datos a través de varios nodos de servidor y confían en Tapestry para dirigir las solicitudes a las réplicas cercanas.

9.4.2 Moneda de datos en DHT

Aunque los DHT proporcionan soporte básico para la replicación, la consistencia mutua de las réplicas tras las actualizaciones puede verse comprometida debido a la salida de pares de la red o a actualizaciones simultáneas. Ilustremos el problema con un escenario de actualización simple en un DHT típico.

Ejemplo 9.7 Supongamos que la operación $\text{put}(k, d_0)$ (emitida por un par) se asigna a los pares p_1 y p_2 , los cuales almacenan los datos d_0 . Consideremos ahora una actualización (del mismo par o de otro) con la operación $\text{put}(k, d_1)$ que también se asigna a los pares p_1 y p_2 . Suponiendo que no se puede acceder a p_2 (por ejemplo, porque ha abandonado la red), solo p_1 se actualiza para almacenar d_1 . Cuando p_2 se reincorpora a la red posteriormente, las réplicas no son consistentes: p_1 contiene el estado actual de los datos asociados con k , mientras que p_2 contiene un estado obsoleto.

Las actualizaciones simultáneas también causan problemas. Consideremos ahora dos actualizaciones $\text{put}(k, d_2)$ y $\text{put}(k, d_3)$ (emitidas por dos pares diferentes) que se envían a p_1 y p_2 en orden inverso, de modo que el último estado de p_1 es d_2 , mientras que el de p_2 es d_3 . Por lo tanto, una operación $\text{get}(k)$ posterior devolverá datos obsoletos o actuales, dependiendo del par consultado, y no hay forma de saber si están actualizados.

Para algunas aplicaciones (p. ej., gestión de agendas, tablones de anuncios, gestión de subastas cooperativas, gestión de reservas, etc.) que podrían aprovechar una DHT, la capacidad de obtener los datos actualizados es fundamental. Para que los datos se mantengan actualizados en las DHT replicadas, es necesario devolver una réplica actualizada incluso si los pares abandonan la red o se producen actualizaciones simultáneas. Por supuesto, la consistencia de las réplicas es un problema más general, como se explica en el capítulo 6, pero este problema es especialmente complejo e importante en los sistemas P2P, dado el considerable dinamismo que existe entre los pares que entran y salen del sistema.

Se ha propuesto una solución que considera tanto la disponibilidad como la vigencia de los datos. Para proporcionar una alta disponibilidad, estos se replican en la DHT mediante un conjunto de funciones hash independientes H_r , denominadas funciones hash de replicación. El par responsable de la clave k con respecto a la función hash h en el momento actual se denota por $\text{rsp}(k, h)$. Para recuperar una réplica actual, cada par (k, datos) se marca con una marca de tiempo lógica, y para cada $h \in H_r$, el par $(k, \text{nuevosDatos})$ se replica en $\text{rsp}(k, h)$, donde $\text{nuevosDatos} = \{\text{datos}, \text{marca de tiempo}\}$; es decir, nuevosDatos se compone de los datos iniciales y la marca de tiempo. Al solicitar los datos asociados a una clave, podemos devolver una de las réplicas con la marca de tiempo más reciente. El número de funciones hash de replicación, es decir, H_r , puede variar según la DHT.

Por ejemplo, si en un DHT la disponibilidad de pares es baja, se puede utilizar un valor alto de H_r (por ejemplo, 30) para aumentar la disponibilidad de datos.

Esta solución es la base de un servicio denominado Servicio de Gestión de Actualizaciones (UMS), que gestiona la inserción y recuperación eficiente de réplicas actuales mediante el sellado de tiempo. La validación experimental ha demostrado que UMS genera muy pocos gastos generales en términos de costes de comunicación. Tras recuperar una réplica, UMS detecta...

Si está vigente o no, es decir, sin tener que comparar con las demás réplicas, y la devuelve como salida. Por lo tanto, UMS no necesita recuperar todas las réplicas para encontrar una vigente; solo requiere el servicio de búsqueda de DHT con operaciones de colocación y obtención .

Para generar marcas de tiempo, UMS utiliza un servicio distribuido llamado Key-based Timestamping Service (KTS). La operación principal de KTS es gen_ts(k), que, dada una clave k , genera un número real como marca de tiempo para k . Las marcas de tiempo generadas por KTS son monótonas , de modo que si ts_i y ts_j son dos marcas de tiempo generadas para la misma clave en los tiempos t_i y t_j , respectivamente, $ts_j > ts_i$ si t_j es posterior a t_i . Esta propiedad permite ordenar las marcas de tiempo generadas para la misma clave según el momento en que se han generado. KTS tiene otra operación denotada por last_ts(k), que, dada una clave k , devuelve la última marca de tiempo generada para k por KTS. En cualquier momento, gen_ts(k) genera como máximo una marca de tiempo para k , y diferentes marcas de tiempo para k son monótonas. Así, en el caso de llamadas concurrentes para insertar un par (k , datos), es decir, de diferentes pares, sólo aquel que obtenga el último timestamp logrará almacenar sus datos en el DHT.

9.4.3 Conciliación de réplicas

La conciliación de réplicas va un paso más allá de la moneda de datos al garantizar la coherencia mutua de las réplicas. Dado que una red P2P suele ser muy dinámica, con pares que se unen o abandonan a voluntad, las soluciones de replicación diligente (véase el capítulo 6) no son adecuadas; se prefiere la replicación diferida. En esta sección, describimos las técnicas de conciliación utilizadas en OceanStore, P-Grid y APPA para ofrecer diversas soluciones.

9.4.3.1 OceanStore

OceanStore es un sistema de gestión de datos diseñado para proporcionar acceso continuo a información persistente. Se basa en Tapestry y asume una infraestructura compuesta por servidores potentes no confiables conectados mediante enlaces de alta velocidad. Por razones de seguridad, los datos se protegen mediante redundancia y técnicas criptográficas. Para mejorar el rendimiento, se permite el almacenamiento en caché de los datos en cualquier punto de la red.

OceanStore permite actualizaciones simultáneas en objetos replicados y se basa en la conciliación para garantizar la consistencia de los datos. Un objeto replicado puede tener múltiples réplicas primarias y secundarias en diferentes nodos. Las réplicas primarias están todas vinculadas y cooperan entre sí para lograr la consistencia mutua de las réplicas mediante el orden de las actualizaciones. Las réplicas secundarias proporcionan un menor grado de consistencia para mejorar el rendimiento y la disponibilidad. Por lo tanto, las réplicas secundarias pueden estar menos actualizadas y ser más numerosas que las primarias. Las réplicas secundarias se comunican entre sí y con las primarias mediante un algoritmo epidémico.

La Figura 9.14 ilustra la gestión de actualizaciones en OceanStore. En este ejemplo, R es el (único) objeto replicado, mientras que R y R_{sec} denotan, respectivamente, un objeto principal y un objeto secundario.

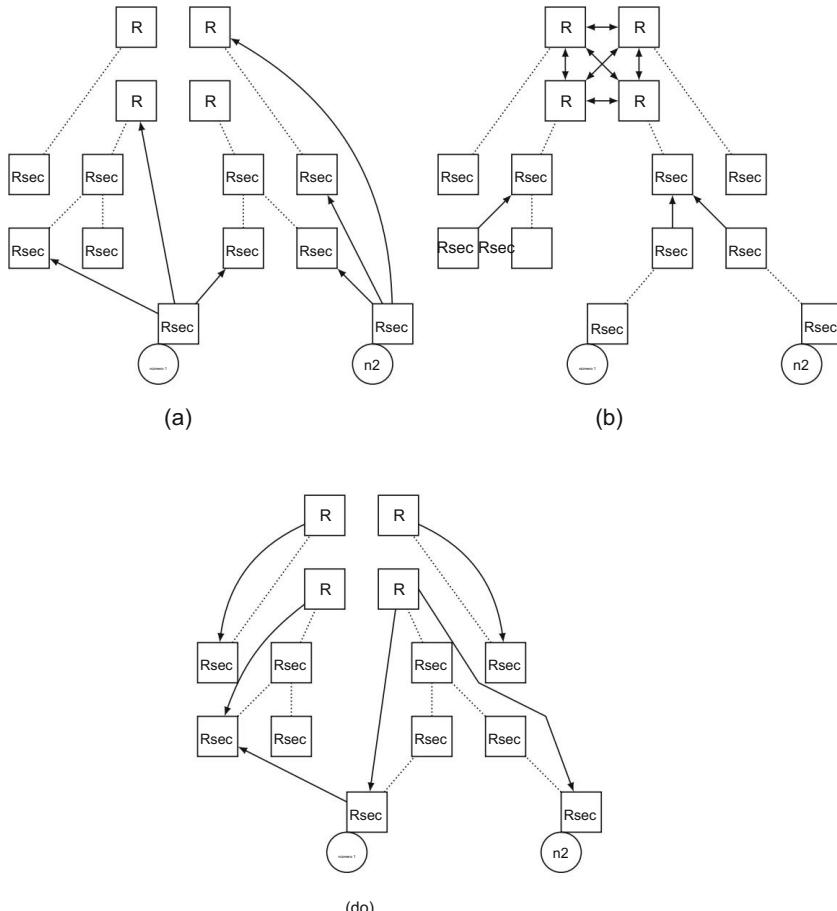


Fig. 9.14 Conciliación de OceanStore. (a) Los nodos n_1 y n_2 envían actualizaciones al grupo maestro de R y a varias réplicas secundarias aleatorias. (b) El grupo maestro de R órdenes se actualiza mientras Las réplicas secundarias las propagan epidémicamente. (c) Despues del acuerdo del grupo maestro, el resultado de actualizaciones se multidifunde a réplicas secundarias

una copia secundaria de R . Los cuatro nodos que contienen una copia primaria están vinculados entre sí. Otros (no se muestran en la figura). Las líneas punteadas representan enlaces entre nodos que contienen Réplicas primarias o secundarias. Los nodos n_1 y n_2 actualizan R simultáneamente.

Las actualizaciones se gestionan de la siguiente manera. Los nodos que contienen copias primarias de R , llamados El grupo maestro es responsable de ordenar las actualizaciones. Por lo tanto, n_1 y n_2 realizan... actualizaciones tentativas sobre sus réplicas secundarias locales y enviar estas actualizaciones a la grupo maestro de R así como a otras réplicas secundarias aleatorias (ver Figura 9.14a).

Las actualizaciones tentativas son ordenadas por el grupo maestro en función de las marcas de tiempo asignadas por n_1 y n_2 ; al mismo tiempo, estas actualizaciones se propagan epidémicamente entre réplicas secundarias (Fig. 9.14b). Una vez que el grupo maestro obtiene un acuerdo, el

El resultado de las actualizaciones se envía por multidifusión a réplicas secundarias (Fig. 9.14c), que contienen datos tentativos² y confirmados.

La gestión de réplicas ajusta la cantidad y la ubicación de las réplicas para atender las solicitudes con mayor eficiencia. Al monitorear la carga del sistema, OceanStore detecta cuándo una réplica está saturada y crea réplicas adicionales en nodos cercanos para aliviar la carga. Por otro lado, estas réplicas adicionales se eliminan cuando ya no son necesarias.

9.4.3.2 Cuadrícula P

P-Grid es una red P2P estructurada basada en una estructura de trie binaria. Un proceso descentralizado y autoorganizado construye la infraestructura de enrutamiento de P-Grid, la cual se adapta a una distribución dada de claves de datos almacenadas por los pares. Este proceso gestiona la distribución uniforme de la carga de almacenamiento de datos y la replicación uniforme de los datos para garantizar la disponibilidad.

Para gestionar las actualizaciones de objetos replicados, P-Grid emplea el método de cotilleo, sin garantías sólidas de consistencia. P-Grid asume que la cuasiconsistencia de las réplicas (en lugar de la consistencia total, que es muy difícil de proporcionar en un entorno dinámico) es suficiente.

El esquema de propagación de actualizaciones consta de una fase de inserción y una fase de extracción. Cuando un par p recibe una nueva actualización de un objeto replicado R , la envía a un subconjunto de pares que contienen réplicas de R , quienes, a su vez, la propagan a otros pares que también las contienen, y así sucesivamente. Los pares que se han desconectado y se conectan de nuevo, los que no reciben actualizaciones durante un tiempo prolongado o los que reciben una solicitud de extracción pero no están seguros de tener la última actualización, entran en la fase de extracción para la conciliación. En esta fase se contacta a múltiples pares y se elige al más actualizado entre ellos para proporcionar el contenido del objeto.

9.4.3.3 APPA

APPA ofrece una solución general de replicación distribuida diferida que garantiza la consistencia final de las réplicas. Utiliza el marco de restricción de acciones IceCube para capturar la semántica de la aplicación y resolver conflictos de actualización.

La semántica de la aplicación se describe mediante restricciones entre acciones de actualización. El programador de la aplicación define una acción y representa una operación específica de la aplicación (p. ej., una operación de escritura en un archivo o documento, o una transacción de base de datos). Una restricción es la representación formal de un invariante de la aplicación. Por ejemplo, la restricción `predSucc(a1, a2)` establece un orden causal entre acciones (es decir, la acción $a2$ se ejecuta solo después de que $a1$ se haya ejecutado correctamente); la restricción `mutuallyExclusive(a1, a2)` establece que se pueden ejecutar tanto $a1$ como $a2$. El objetivo de

²Los datos tentativos son datos que las réplicas principales aún no han confirmado.

La conciliación consiste en tomar un conjunto de acciones con las restricciones asociadas y generar un cronograma, es decir, una lista de acciones ordenadas que no las infringen. Para reducir la complejidad de la producción del cronograma, el conjunto de acciones a ordenar se divide en subconjuntos denominados clústeres. Un clúster es un subconjunto de acciones relacionadas por restricciones que pueden ordenarse independientemente de otros clústeres. Por lo tanto, el cronograma global se compone de la concatenación de las acciones ordenadas de los clústeres.

Los datos gestionados por el algoritmo de conciliación APPA se almacenan en estructuras de datos denominadas objetos de conciliación. Cada objeto de conciliación tiene un identificador único que permite su almacenamiento y recuperación en la DHT. La replicación de datos se realiza de la siguiente manera. Primero, los nodos ejecutan acciones locales para actualizar una réplica de un objeto, respetando las restricciones definidas por el usuario. Posteriormente, estas acciones (con las restricciones asociadas) se almacenan en la DHT según el identificador del objeto. Finalmente, los nodos conciliadores recuperan las acciones y restricciones de la DHT y generan la programación global, conciliando las acciones conflictivas según la semántica de la aplicación. Esta programación se ejecuta localmente en cada nodo, garantizando así la consistencia final.

Cualquier nodo conectado puede intentar iniciar la conciliación invitando a otros nodos disponibles a participar. Solo se puede ejecutar una conciliación a la vez. La conciliación de las acciones de actualización se realiza en 6 pasos distribuidos, como se indica a continuación. Los nodos en el paso 2 iniciaron la conciliación. Las salidas generadas en cada paso se convierten en la entrada del siguiente.

- Paso 1: asignación de nodos: se selecciona un subconjunto de nodos de réplica conectados para proceder como conciliadores en función de los costos de comunicación.
- Paso 2: Agrupación de acciones: Los conciliadores extraen acciones de los registros de acciones y agrupan las acciones que intentan actualizar objetos comunes, ya que podrían entrar en conflicto. Los grupos de acciones que intentan actualizar el objeto R se almacenan en el objeto de conciliación (LR) del registro de acciones R.
- Paso 3: creación de clústeres: Los conciliadores extraen grupos de acciones de los registros de acciones y los dividen en clústeres de acciones conflictivas semánticamente dependientes: dos acciones, a1 y a2, son semánticamente independientes si la aplicación considera seguro ejecutarlas juntas, en cualquier orden, incluso si actualizan un objeto común; de lo contrario, a1 y a2 son semánticamente dependientes. Los clústeres generados en este paso se almacenan en el objeto de conciliación del conjunto de clústeres.
- Paso 4: Extensión de clústeres: Las restricciones definidas por el usuario no se tienen en cuenta al crear clústeres. Por lo tanto, en este paso, los conciliadores extienden los clústeres añadiéndoles nuevas acciones conflictivas, según las restricciones definidas por el usuario. •
- Paso 5: Integración de clústeres: Las extensiones de clústeres provocan superposición de clústeres (una superposición se produce cuando la intersección de dos clústeres da como resultado un conjunto de acciones no nulo). En este paso, los conciliadores unen los clústeres superpuestos. En este punto, los clústeres se vuelven mutuamente independientes, es decir, no hay restricciones que involucren acciones de clústeres
- distintos. • Paso 6: Ordenación de clústeres: En este paso, los conciliadores toman cada clúster del conjunto de clústeres y ordenan sus acciones. Las acciones ordenadas asociadas a cada clúster se almacenan en el objeto de conciliación de la programación . La concatenación

Las acciones ordenadas de todos los clústeres conforman la programación global que ejecutan todos los nodos de réplica.

En cada paso, el algoritmo de reconciliación aprovecha el paralelismo de datos, es decir, varios nodos realizan simultáneamente actividades independientes en un subconjunto distinto de acciones (por ejemplo, ordenamiento de diferentes clústeres).

9.5 Cadena de bloques

Popularizada por bitcoin y otras criptomonedas, blockchain es una infraestructura P2P reciente que puede registrar transacciones entre dos partes de manera eficiente y segura.

Se ha convertido en un tema candente, sujeto a mucha controversia y controversia. Por un lado, encontramos defensores entusiastas como Ito, Narula y Ali, quienes en 2017 afirmaron que blockchain es una tecnología disruptiva que "le hará al sistema financiero lo que internet le hizo a los medios". Por otro lado, encontramos fuertes oponentes, como el famoso economista N. Roubini, quien en 2018 calificó a blockchain como la tecnología más "sobrevalorada y menos útil de la historia de la humanidad". Como siempre, la verdad probablemente se encuentre en un punto intermedio.

La blockchain se inventó para Bitcoin con el fin de resolver el problema del doble gasto de las monedas digitales anteriores sin la necesidad de una autoridad central de confianza. El 3 de enero de 2009, Satoshi Nakamoto³ creó el primer bloque fuente con una transacción única de 50 bitcoins para sí mismo. Desde entonces, han surgido muchas otras blockchains, como Ethereum en 2013 y Ripple en 2014. El éxito ha sido considerable y las criptomonedas se han utilizado ampliamente para transferencias de dinero o inversiones de alto riesgo, por ejemplo, las ofertas iniciales de monedas (ICO) como alternativa a las ofertas públicas iniciales (IPO). Las posibles ventajas de usar una criptomoneda basada en blockchain son las siguientes:

- Tarifa de transacción baja (establecida por el remitente para acelerar el procesamiento), que es independiente de la cantidad de dinero transferido; • Menos riesgos para los comerciantes (sin devoluciones de cargos fraudulentos);
- Seguridad y control (por ejemplo, protección contra el robo de identidad);
- Confianza a través de la cadena de bloques, sin ninguna autoridad central.

Sin embargo, las criptomonedas también se han utilizado con frecuencia para estafas y actividades ilegales (compras en la red oscura, blanqueo de capitales, robo, etc.), lo que ha generado advertencias por parte de las autoridades del mercado y el inicio de su regulación en algunos países. Otros problemas son:

- inestable: ya que no hay respaldo de un banco estatal o federal (a diferencia de los bancos fuertes) monedas como el dólar o el euro);

³Pseudo para la persona o personas que desarrollaron bitcoin, lo que generó mucha especulación sobre su verdadera identidad.

- no relacionado con la economía real, lo que fomenta la especulación; • altamente volátil, por ejemplo, el tipo de cambio con una moneda real (tal como lo establecen los mercados de criptomonedas) puede variar mucho en unas pocas horas;
- sujeto a graves estallidos de burbujas criptográficas, como en 2017.

Por lo tanto, las criptomonedas basadas en blockchain tienen sus ventajas y desventajas. Sin embargo, no debemos limitar la blockchain a las criptomonedas, ya que existen muchas otras aplicaciones útiles. La blockchain original es un libro de contabilidad público y distribuido que puede registrar y compartir transacciones entre varias computadoras de forma segura y permanente. Es una infraestructura compleja de base de datos distribuida que combina diversas tecnologías como P2P, replicación de datos, consenso y cifrado de clave pública.

El término Blockchain 2.0 se refiere a nuevas aplicaciones que pueden programarse en la cadena de bloques para ir más allá de las transacciones y permitir el intercambio de activos sin intermediarios poderosos. Ejemplos de estas aplicaciones son los contratos inteligentes, las identificaciones digitales persistentes, los derechos de propiedad intelectual, los blogs, las votaciones, la reputación, etc.

9.5.1 Definición de blockchain

El registro de transacciones financieras entre dos partes se ha realizado tradicionalmente mediante un libro de contabilidad centralizado intermedio, es decir, una base de datos de todas las transacciones, controlada por una autoridad de confianza, por ejemplo, una cámara de compensación. En un mundo digital, este enfoque centralizado presenta varios problemas. En primer lugar, crea puntos únicos de fallo y lo convierte en un blanco atractivo para los atacantes. En segundo lugar, favorece la concentración de actores, como las grandes instituciones financieras. En tercer lugar, las transacciones complejas que requieren múltiples intermediarios, generalmente con sistemas y reglas heterogéneos, pueden ser difíciles y su ejecución requiere tiempo.

Una cadena de bloques (blockchain) es esencialmente un libro de contabilidad distribuido compartido entre varios nodos participantes en una red P2P. Está organizada como una base de datos de bloques replicada y de solo anexión. Los bloques son contenedores digitales para transacciones y están protegidos mediante cifrado de clave pública. El código de cada nuevo bloque se basa en el del bloque anterior, lo que garantiza su inviolabilidad. La cadena de bloques es vista por todos los participantes que mantienen copias de la base de datos en modo multamaestro (véase el capítulo 6) y colaboran por consenso en la validación de las transacciones en los bloques. Una vez validada y registrada en un bloque, una transacción no se puede modificar ni eliminar, lo que hace que la cadena de bloques sea a prueba de manipulaciones. Los nodos participantes pueden no confiar plenamente entre sí y algunos incluso pueden comportarse de forma maliciosa (bizantina), es decir, asignar valores diferentes a distintos nodos observadores. Por lo tanto, en el caso general, es decir, una cadena de bloques pública como Bitcoin, la cadena de bloques debe tolerar fallos bizantinos.

Tenga en cuenta que el objetivo de una estructura de datos P2P típica, como una DHT, es proporcionar una búsqueda rápida y escalable. El propósito de una cadena de bloques es muy diferente: gestionar una lista de bloques en constante crecimiento de forma segura y a prueba de manipulaciones.

Pero la escalabilidad no es un objetivo ya que la cadena de bloques no está dividida en nodos P2P.

En comparación con el enfoque de libro mayor centralizado, la cadena de bloques puede traer las siguientes ventajas:

- Mayor confianza en las transacciones y el intercambio de valor, al confiar en los datos, no en los participantes.
 - Mayor fiabilidad (sin punto único de fallo) gracias a la replicación. • Seguridad integrada mediante el encadenamiento de bloques y el cifrado de clave pública. • Transacciones eficientes y más económicas entre participantes, en particular, en comparación con...
- con la dependencia de una larga cadena de intermediarios.

Las cadenas de bloques se pueden utilizar en dos tipos de mercados: público (por ejemplo, criptomonedas, subasta pública), donde cualquiera puede participar; y privado (por ejemplo, gestión de la cadena de suministro, atención médica), donde los participantes son conocidos. Por lo tanto, es importante distinguir entre cadenas de bloques públicas y privadas (también llamadas permisionadas).

Una cadena de bloques pública (como Bitcoin) es una red P2P abierta, sin permisos, y puede ser de gran escala. Los participantes son desconocidos y poco confiables, y pueden unirse y abandonar la red sin previo aviso. Suelen estar pseudonimizados, lo que permite rastrear todo el historial de transacciones de un participante e incluso, en ocasiones, identificarlo.

Una blockchain privada es una red cerrada y con permisos, por lo que su escala suele ser mucho menor que la de una blockchain pública. El control está regulado para garantizar que solo los participantes identificados y autorizados puedan validar las transacciones. El acceso a las transacciones de la blockchain puede restringirse a los participantes autorizados, lo que aumenta la protección de datos. Aunque la infraestructura subyacente puede ser la misma, la principal diferencia entre una blockchain pública y una privada radica en quién (persona, grupo o empresa) puede participar en la red y quién la controla.

9.5.2 Infraestructura de cadena de bloques

En esta sección, presentamos la infraestructura de la cadena de bloques tal como se propuso originalmente para Bitcoin, centrandonos en el proceso de procesamiento de transacciones. Los nodos participantes se denominan nodos completos para distinguirlos de otros nodos, como los nodos cliente ligeros que gestionan billeteras digitales. Cuando un nuevo nodo completo se une a la red, se sincroniza con los nodos conocidos mediante el Sistema de Nombres de Dominio (DNS) para obtener una copia de la cadena de bloques. Posteriormente, puede crear transacciones y convertirse en un "minero", es decir, participar en la validación de bloques, lo que se conoce como proceso de "minería".

El procesamiento de transacciones se realiza en tres pasos principales:

1. Crear una transacción después de que dos usuarios hayan acordado la información de la transacción Intercambio: direcciones de billetera, claves públicas, etc.
2. Agrupación de transacciones en un bloque y vinculación con un bloque anterior.
3. Validación del bloque (y de las transacciones) mediante "minería", adición de la bloque validado en la blockchain y replicación en la red.

En el resto de esta sección, presentamos cada paso con más detalle.

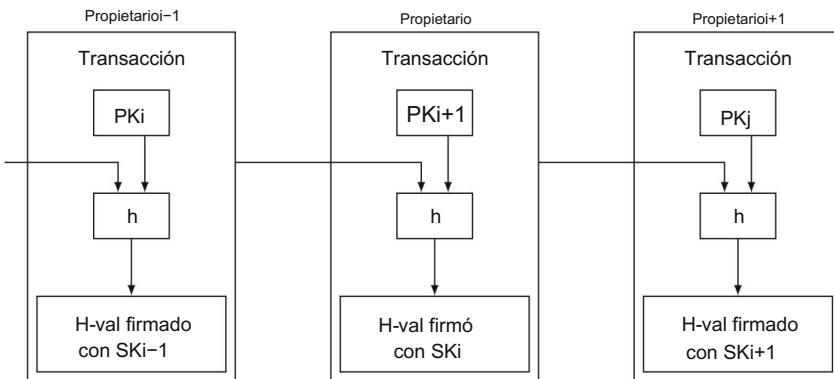


Fig. 9.15 Encadenamiento de transacciones

9.5.2.1 Creación de una transacción

Consideremos una transacción de bitcoin entre el propietario de una moneda y el receptor de la misma. La transacción está protegida mediante cifrado de clave pública y firma digital. Cada propietario tiene una clave pública y una privada. El propietario de la moneda firma la transacción mediante...

- crear un resumen hash de una combinación de la transacción anterior (con la que recibe las monedas) y de la clave pública del siguiente propietario;
- firmar el resumen hash con su clave privada.

Esta firma se añade al final de la transacción, creando así una cadena de transacciones entre todos los propietarios (véase la Fig. 9.15). Posteriormente, el propietario de la moneda publica la transacción en la red mediante multidifusión a todos los demás nodos. Dada la clave pública del propietario de la moneda que creó la transacción, cualquier nodo de la red puede verificar la firma de la transacción.

9.5.2.2 Agrupación de transacciones en bloques

El doble gasto es una posible falla en un sistema de dinero digital, ya que un mismo token digital puede gastarse más de una vez. A diferencia del dinero físico, un token digital consiste en un archivo digital que puede duplicarse o falsificarse.

Cada nodo minero (que mantiene una copia de la cadena de bloques) recibe las transacciones publicadas, las valida y las agrupa en bloques. Para aceptar una transacción e incluirla en un bloque, los mineros siguen ciertas reglas, como verificar que las entradas sean válidas y que una moneda no se gaste dos veces (más de una vez) como resultado de un ataque (véase el ataque del 51% a continuación). Es posible que un minero malicioso intente aceptar una transacción que infrinja algunas reglas e incluirla en un bloque. En este caso, el bloque no obtendrá el consenso de otros mineros.

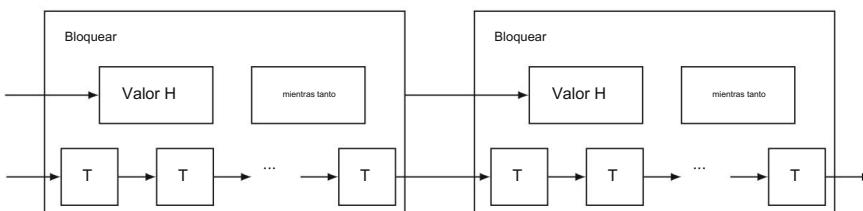


Fig. 9.16 Encadenamiento de bloques

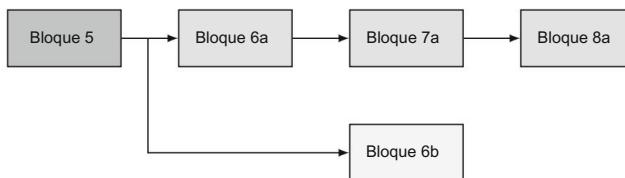


Fig. 9.17 Regla de la cadena más larga

que siguen las reglas y no serán aceptados ni incluidos en la cadena de bloques. Por lo tanto,

Si la mayoría de los mineros siguen la regla, el sistema funciona. Como se muestra en la figura 9.16,

Cada nuevo bloque se construye sobre un bloque anterior de la cadena produciendo un hash.

resumen (valor h) de la dirección del bloque anterior, protegiendo así al bloque de manipulación o cambio. El tamaño actual de un bloque de bitcoin es de 1 megabyte, lo que refleja...

Compromiso entre eficiencia y seguridad.

Un problema que puede surgir es una bifurcación accidental o intencional. Como diferentes bloques... se validan en paralelo por diferentes nodos, un nodo puede ver varios candidatos

cadenas en cualquier momento. Por ejemplo, en la figura 9.17, un nodo puede ver los bloques 7a y 6b,

Ambos se originaron en el bloque 5. La solución es aplicar la regla de la cadena más larga, es decir,

Elija el bloque que se encuentra en la cadena más larga. En el ejemplo de la Fig. 9.17, el

Se elegirá el bloque 7a para construir el siguiente bloque 7b. La razón de esta regla es...

Para minimizar la cantidad de transacciones que deben reenviarse. Por ejemplo,

Las transacciones en el Bloque 6b deben ser reenviadas por el cliente (quien verá que

El bloque no ha sido validado). Por lo tanto, las transacciones en un bloque validado solo son

Validado provisionalmente y se debe esperar confirmación. Cada nuevo bloque aceptado

en la cadena después de la validación de la transacción se considera como una confirmación.

Bitcoin considera que una transacción está madura después de 6 confirmaciones (1 hora en promedio). En

Fig. 9.17, el vencimiento de la transacción se ilustra mediante la oscuridad de los cuadros (bloque 6b)

es más ligero porque sus transacciones no serán confirmadas).

Además de las bifurcaciones accidentales, también existen las bifurcaciones intencionales, que son útiles

para agregar nuevas características a la base del código de blockchain (cambios de protocolo) o revertir

Los efectos de hacks o errores catastróficos. Hay dos tipos de bifurcación posibles: bifurcación suave

versus bifurcación dura. Una bifurcación suave es compatible con versiones anteriores: el software antiguo reconoce

Los bloques creados con nuevas reglas son válidos. Sin embargo, esto facilita el acceso a los atacantes.

El caso más famoso de una bifurcación dura es el de la cadena de bloques Ethereum en 2016, después de...

Un ataque contra un contrato inteligente complejo para capital de riesgo. Ethereum se bifurcó, pero sin el impulso de la comunidad que gestionaba el software, lo que dio lugar a dos cadenas de bloques: la (nueva) Ethereum y la (antigua) Ethereum Classic. Cabe destacar que la batalla ha sido más filosófica y ética que técnica.

9.5.2.3 Validación de bloques por consenso

Dado que los bloques se producen en paralelo por nodos que compiten, se necesita un consenso para validarlos y añadirlos a la cadena de bloques. Cabe destacar que, en el caso general de la cadena de bloques pública, donde los participantes son desconocidos, los protocolos de consenso tradicionales como Paxos (véase la sección 5.4.5) no son aplicables. El protocolo de consenso de la cadena de bloques de Bitcoin se basa en la minería. ⁴ Podemos resumir el protocolo de consenso de la siguiente manera:

1. Los nodos mineros compiten (como en una lotería) para producir nuevos bloques. Con un alto consumo de potencia computacional, cada minero intenta producir un nonce (número de un solo uso) para el bloque (véase la Fig. 9.16).
2. Una vez que un minero ha encontrado el nonce, agrega el bloque a la cadena de bloques y lo transmite en multidifusión a todos los nodos de la red.
3. Otros nodos verifican el nuevo bloque, comprobando el nonce (lo cual es fácil).
4. Dado que muchos nodos intentan ser los primeros en agregar un bloque a la cadena de bloques, un sistema de recompensa basado en lotería selecciona uno de los bloques en competencia, basándose en cierta probabilidad, y el ganador recibe el pago, por ejemplo, 12,5 bitcoins hoy (originalmente 50). Esto aumenta la oferta monetaria.

La minería está diseñada para ser difícil. Cuanto mayor sea la potencia de minería de la red, más difícil será calcular el nonce. Esto permite controlar la inyección de nuevos bloques ("inflación") en el sistema, con un promedio de un bloque cada 10 minutos. La dificultad de la minería consiste en generar una Prueba de Trabajo (PoW), es decir, un dato difícil de calcular pero fácil de verificar, para calcular el nonce. PoW se propuso inicialmente para prevenir ataques DoS. La blockchain de Bitcoin utiliza Hashcash PoW, basado en la función hash SHA-256. El objetivo es generar un valor v tal que $h(f(\text{bloque}, v)) < T$

, donde

1. h es la función hash SHA-256; 2. f es una función que combina v con información en el bloque, por lo que el nonce no puede ser precalculado; 3.

T es un valor objetivo compartido por todos los nodos y refleja el tamaño de la red; 4. v es un número de 256 bits que comienza con n bits cero.

El esfuerzo promedio para producir el PoW es exponencial en la cantidad de bits cero requeridos, es decir, la probabilidad de éxito es baja y puede aproximarse como $1/2^n$.

El término se utiliza por analogía con la minería de oro como el proceso de sacar a la luz monedas que existen en el diseño del protocolo.

Esto beneficia a los nodos potentes, que ahora utilizan grandes clústeres de GPU. Sin embargo, la verificación es muy sencilla y puede realizarse mediante la ejecución de una sola función hash.

Un problema potencial con la minería basada en PoW es el ataque del 51%, que permite al atacante invalidar transacciones válidas y duplicar el gasto de fondos. Para ello, el atacante (un minero o una coalición de mineros) debe poseer más del 50% de la potencia computacional total para la minería.

Entonces, es posible modificar una cadena recibida (por ejemplo, eliminando una transacción) y generar una cadena más larga que será seleccionada por la mayoría según la regla de la cadena más larga.

9.5.3 Blockchain 2.0

La cadena de bloques de primera generación, iniciada por Bitcoin, permite el registro de transacciones y el intercambio de criptomonedas sin intermediarios poderosos.

Blockchain 2.0 es una evolución importante del paradigma que trasciende las transacciones y permite el intercambio de todo tipo de activos. Impulsado por Ethereum, permite que la blockchain sea programable, permitiendo a los desarrolladores de aplicaciones crear API y servicios directamente en ella.

Las características esenciales de las aplicaciones son el intercambio de activos y valor (mediante transacciones), la participación de múltiples participantes, posiblemente desconocidos entre sí, y la confianza (en los datos) es crucial. Blockchain 2.0 tiene numerosas aplicaciones en diversos sectores, como servicios financieros y micropagos, derechos digitales, gestión de la cadena de suministro, registro sanitario, Internet de las Cosas (IoT) y procedencia de alimentos. La mayoría de estas aplicaciones pueden funcionar con una blockchain privada. En este caso, las principales ventajas son una mayor privacidad y control, y una validación de transacciones más eficiente, ya que los participantes son confiables y no es necesario generar una prueba de trabajo (PoW).

Una capacidad importante que Blockchain 2.0 puede soportar son los contratos inteligentes. Un contrato inteligente es un contrato autoejecutable, con código que integra los términos y condiciones del contrato. Un ejemplo de contrato inteligente simple es un contrato de servicio entre dos partes: una que solicita el servicio con un pago asociado y la otra que lo ejecuta y, una vez ejecutado, recibe el pago. En una blockchain, los contratos pueden ejecutarse parcial o totalmente sin interacción humana e involucrar a muchos participantes, por ejemplo, dispositivos IoT. Una gran ventaja de tener contratos inteligentes en la blockchain es que el código, que implementa el contrato, se vuelve visible para todos para su verificación. Sin embargo, una vez en la blockchain, el contrato no se puede modificar. Desde un punto de vista técnico, el principal desafío es producir código libre de errores, lo cual se lograría mejor mediante la verificación de código.

Una importante iniciativa colaborativa para producir cadenas de bloques de código abierto y herramientas relacionadas es el proyecto Hyperledger de la Fundación Linux, iniciado en 2015 por IBM, Intel, Cisco y otros. Los principales marcos de trabajo son:

- Hyperledger Fabric (IBM, activo digital): una infraestructura de cadena de bloques permissionada con contratos inteligentes, consenso configurable y servicios de membresía.

- Sawtooth (Intel): un novedoso mecanismo de consenso, "Prueba de tiempo transcurrido", que Se basa en entornos de ejecución confiables.
- Hyperledger Iroha (Soramitsu): basado en Hyperledger Fabric, con un enfoque en aplicaciones móviles.

9.5.4 Problemas

La cadena de bloques se promociona a menudo como una tecnología disruptiva para registrar transacciones y verificar registros, con un gran impacto en el sector financiero. En particular, la capacidad de programar aplicaciones y lógica de negocio en la cadena de bloques abre numerosas posibilidades para los desarrolladores, como por ejemplo, los contratos inteligentes. Algunos defensores, como los activistas cypherpunk, incluso la consideran un potencial poder disruptivo que establecerá un sentimiento de democracia e igualdad, donde las personas y las pequeñas empresas podrán competir con el poder corporativo.

Sin embargo, existen limitaciones importantes, en particular en el caso del sector público.

Blockchain, como infraestructura más general, presenta las siguientes limitaciones:

- Complejidad y escalabilidad, en particular, difícil evolución de las reglas operativas que requieren bifurcar la cadena de bloques.
- Cadenas cada vez más grandes y alto consumo de energía (con PoW). • Potencial de ataque del 51%. • Baja privacidad, ya que los usuarios solo están seudonimizados. Por ejemplo, al realizar una transacción. con un usuario puede revelar todas sus otras transacciones.
- Duración impredecible de las transacciones, desde unos pocos minutos hasta días. • Falta de control y regulación, lo que dificulta la vigilancia y la imposición de impuestos por parte de los estados. actas.
- Preocupaciones de seguridad: si se pierde o se roba una clave privada, una persona no tiene ningún recurso.

Para abordar estas limitaciones, se pueden identificar varios problemas de investigación en sistemas distribuidos, ingeniería de software y gestión de datos:

- Escalabilidad y seguridad de la blockchain pública. Este problema ha renovado el interés en los protocolos de consenso, con alternativas más eficientes a PoW: prueba de participación, prueba de retención, prueba de uso y prueba de participación/tiempo. Además, existen otros cuellos de botella en el rendimiento, además del consenso. Sin embargo, un problema importante sigue siendo el equilibrio entre rendimiento y seguridad. Bitcoin-NG es una blockchain de nueva generación con dos tipos de bloques: bloques clave que incluyen PoW, una referencia al bloque anterior y la recompensa de minería, lo que hace que la computación PoW sea más eficiente; y microbloques que incluyen transacciones, pero no PoW. • Gestión de contratos inteligentes, incluyendo la certificación y verificación de código, la evolución de contratos (propagación de cambios), la optimización y el control de ejecución. • Blockchain y gestión de datos. Dado que una blockchain es simplemente una estructura de base de datos distribuida, puede mejorarse basándose en los principios de diseño de los sistemas de bases de datos. Por ejemplo, un lenguaje declarativo podría facilitarla.

Para definir, verificar y optimizar contratos inteligentes complejos. BigchainDB es un nuevo sistema de gestión de bases de datos (SGBD) que aplica conceptos de bases de datos distribuidas, en particular, un modelo de transacciones enriquecido, control de acceso basado en roles y consultas, para respaldar una cadena de bloques escalable. Comprender los cuellos de botella en el rendimiento también requiere análisis comparativos. BLOCKBENCH es un marco de referencia para comprender el rendimiento de las cadenas de bloques privadas frente a las cargas de trabajo de procesamiento de datos. •

Interoperabilidad de cadenas de bloques. Existen numerosas cadenas de bloques, cada una con diferentes protocolos y API. La Alianza de Interoperabilidad de Cadenas de Bloques (BIA) se creó para definir estándares que promuevan las transacciones entre cadenas de bloques.

9.6 Conclusión

Al distribuir el almacenamiento y el procesamiento de datos entre pares autónomos en la red, los sistemas P2P pueden escalar sin necesidad de servidores potentes. Hoy en día, millones de usuarios utilizan a diario importantes aplicaciones de intercambio de datos como BitTorrent, eDonkey o Gnutella. El P2P también se ha utilizado con éxito para escalar la gestión de datos en la nube, por ejemplo, el almacén de clave-valor de DynamoDB (véase la sección 11.2.1). Sin embargo, estas aplicaciones siguen presentando limitaciones en cuanto a la funcionalidad de la base de datos.

Las aplicaciones P2P avanzadas, como el consumo colaborativo (p. ej., el uso compartido de vehículos), deben gestionar datos semánticamente ricos (p. ej., documentos XML o RDF, tablas relacionales, etc.). Para dar soporte a estas aplicaciones, es necesario revisar considerablemente las técnicas de bases de datos distribuidas (gestión de esquemas, control de acceso, procesamiento de consultas, gestión de transacciones, gestión de la consistencia, fiabilidad y replicación).

Al considerar la gestión de datos, los principales requisitos de un sistema de gestión de datos P2P son la autonomía, la expresividad de las consultas, la eficiencia, la calidad del servicio y la tolerancia a fallos.

Dependiendo de la arquitectura de la red P2P (no estructurada, estructurada DHT o superpeer), estos requisitos pueden alcanzarse en distintos grados.

Las redes no estructuradas ofrecen mayor tolerancia a fallos, pero pueden ser bastante ineficientes debido a que dependen de la inundación para el enrutamiento de consultas. Los sistemas híbridos tienen mayor potencial para satisfacer requisitos de gestión de datos de alto nivel. Sin embargo, los sistemas DHT son más adecuados para la búsqueda basada en claves y podrían combinarse con redes superpeer para búsquedas más complejas.

La mayor parte del trabajo sobre intercambio de datos en sistemas P2P se ha centrado inicialmente en la gestión de esquemas y el procesamiento de consultas, en particular para tratar datos semánticamente ricos. Sin embargo, más recientemente, con la tecnología blockchain, se ha profundizado mucho en la gestión de actualizaciones, la replicación, las transacciones y el control de acceso, aunque con datos relativamente simples. Las técnicas P2P también han recibido atención para facilitar la escalabilidad de la gestión de datos en el contexto de la computación en red o para proteger la privacidad de los datos en la recuperación de información o el análisis de datos.

La investigación sobre la gestión de datos P2P está despertando un renovado interés en dos contextos principales: blockchain y edge computing. En el contexto de blockchain, los principales temas de investigación, que analizamos en profundidad al final de la sección 9.5, se relacionan con la escalabilidad y la seguridad de la blockchain pública (p. ej., protocolos de consenso), la inteligencia...

Gestión de contratos, en particular mediante lenguajes de consulta declarativos, benchmarking e interoperabilidad blockchain. En el contexto de la computación de borde, generalmente con dispositivos IoT, los servidores móviles de borde podrían organizarse como una red P2P para descargar las tareas de gestión de datos. Por lo tanto, la problemática se encuentra en la intersección entre la computación móvil y la P2P.

9.7 Notas bibliográficas

La gestión de datos en los sistemas P2P modernos se caracteriza por su distribución masiva, heterogeneidad inherente y alta volatilidad. El tema se aborda en profundidad en varios libros, incluyendo [Vu et al. 2009, Pacitti et al. 2012]. Un análisis más breve se puede encontrar en [Ulusoy 2007]. Se ofrecen análisis sobre los requisitos, las arquitecturas y los problemas que enfrentan los sistemas de gestión de datos P2P en [Bernstein et al. 2002, Daswani et al. 2003, Valduriez y Pacitti 2004]. Se presentan varios sistemas de gestión de datos P2P en [Aberer 2003].

En redes P2P no estructuradas, el problema de la inundación se gestiona mediante uno de los dos métodos indicados. La selección de un subconjunto de vecinos para reenviar solicitudes se debe a Kalogeraki et al. [2002]. El uso de recorridos aleatorios para elegir el conjunto de vecinos se debe a Lv et al. [2002], el uso de un índice de vecindad dentro de un radio se debe a Yang y García-Molina [2002], y el mantenimiento de un índice de recursos para determinar la lista de vecinos con mayor probabilidad de estar en la dirección del par buscado se debe a Crespo y García-Molina [2002]. La propuesta alternativa de utilizar el protocolo epidémico se analiza en [Kermarrec y van Steen 2007], basándose en el chisme que se analiza en [Demers et al. 1987]. Los enfoques para escalar el chisme se proporcionan en [Voulgaris et al. 2003].

Las redes P2P estructuradas se discuten en [Ritter 2001, Ratnasamy et al. 2001, Stoica et al. 2001]. Similar a las DHT, el hash dinámico también se ha usado con éxito para abordar los problemas de escalabilidad de estructuras de archivos distribuidos muy grandes [Devine 1993, Litwin et al. 1993]. Las superposiciones basadas en DHT se pueden categorizar según su geometría de enrutamiento y algoritmo de enrutamiento [Gummadi et al. 2003]. Presentamos con más detalle las siguientes DHT: Tapestry [Zhao et al. 2004], CAN [Ratnasamy et al. 2001] y Chord [Stoica et al. 2003]. Las redes P2P estructuradas jerárquicas que discutimos y sus publicaciones fuente son las siguientes: PHT [Ramabhadran et al. 2004], P-Grid [Aberer 2001, Aberer et al. 2003a], BATON [Jagadish et al. 2005], BATON* [Jagadish et al. 2006], árbol VBI [Jagadish et al.

2005], P-Tree [Crainiceanu et al. 2004], SkipNet [Harvey et al. 2003] y Skip Graph [Aspnes y Shah 2003]. Schmidt y Parashar [2004] describen un sistema que utiliza curvas que llenan el espacio para definir la estructura, y Ganeshan et al. [2004] proponen uno basado en la estructura de hiperrectángulo.

Entre los ejemplos de redes superpares se incluyen Edutella [Nejdl et al. 2003] y JXTA.

Se puede encontrar una buena discusión sobre las cuestiones del mapeo de esquemas en sistemas P2P en [Tatarinov et al. 2003]. El mapeo de esquemas por pares se utiliza en Piazza [Tatarinov et al. 2003], LRM [Bernstein et al. 2002], Hyperion [Kementsietsidis et al. 2003],

y PGrid [Aberer et al., 2003b]. El mapeo basado en técnicas de aprendizaje automático se utiliza en GLUE [Doan et al., 2003b]. El mapeo de acuerdo común se utiliza en APPA [Akbarinia et al., 2006, Akbarinia y Martins, 2007] y AutoMed [McBrien y Poulovassilis , 2003]. El mapeo de esquemas mediante técnicas IR se utiliza en PeerDB [Ooi et al., 2003] y Edutella [Nejdl et al., 2003]. La reformulación de consultas semánticas mediante mapeos de esquemas por pares en sistemas P2P sociales se aborda en [Bonifati et al., 2014].

En [Akbarinia et al. 2007b] se ofrece un estudio exhaustivo del procesamiento de consultas en sistemas P2P y ha sido la base para escribir las Secciones 9.2 y 9.3.

Un tipo importante de consulta en sistemas P2P son las consultas top-k. Un estudio de las técnicas de procesamiento de consultas top-k en sistemas de bases de datos relacionales se proporciona en [Ilyas et al. 2008]. Un algoritmo eficiente para el procesamiento de consultas top-k es el Algoritmo de Umbral (TA), que fue propuesto independientemente por varios investigadores [Nepal y Ramakrishna 1999, Güntzer et al. 2000, Fagin et al. 2003]. TA ha sido la base de varios algoritmos en sistemas P2P, en particular en DHT [Akbarinia et al. 2007a]. Un algoritmo más eficiente que TA es el Algoritmo de Mejor Posición [Akbarinia et al. 2007c]. Se han propuesto varios algoritmos de estilo TA para el procesamiento distribuido de consultas top-k, por ejemplo, TPUT [Cao y Wang 2004].

El procesamiento de consultas top-k en sistemas P2P ha recibido mucha atención: en sistemas no estructurados, p. ej., PlanetP [Cuenca-Acuna et al. 2003] y APPA [Akbarinia et al. 2006]; en DHT, p. ej., APPA [Akbarinia et al. 2007a]; y en sistemas superpeer, p. ej., Edutella [Balke et al. 2005]. Se proponen soluciones para el procesamiento de consultas de unión P2P en PIER [Huebsch et al. 2003]. Se proponen soluciones para el procesamiento de consultas de rango P2P en hashing sensible a la localidad [Gupta et al. 2003], PHT [Ramabhadran et al. 2004] y BATON [Jagadish et al. 2005].

El estudio de la replicación en sistemas P2P realizado por Martins et al. [2006b] ha servido de base para la sección 9.4. En [Akbarinia et al. 2007d] se ofrece una solución completa para la actualización de datos en DHT replicados, es decir, que permite encontrar la réplica más actualizada. La conciliación de datos replicados se aborda en OceanStore [Kubiatowicz et al. 2000], P-Grid [Aberer et al. 2003a] y APPA [Martins et al. 2006a, Martins y Pacitti 2006, Martins et al. 2008]. El marco de acción-restricción se ha propuesto para IceCube [Kermarrec et al. 2001].

Las técnicas P2P también han recibido atención para ayudar a ampliar la gestión de datos en el contexto de la computación en red [Pacitti et al. 2007] o la computación de borde/móvil [Tang et al. 2019], o para ayudar a proteger la privacidad de los datos en el análisis de datos [Allard et al. 2015].

La cadena de bloques (blockchain) es un tema relativamente reciente y polémico, con defensores entusiastas [Ito et al., 2017] y firmes detractores, como el famoso economista N. Roubini [Roubini , 2018]. Los conceptos se definen en el artículo pionero sobre la cadena de bloques de bitcoin [Nakamoto, 2008]. Desde entonces, se han propuesto muchas otras cadenas de bloques para otras criptomonedas, como Ethereum y Ripple. La mayoría de las contribuciones iniciales han sido realizadas por desarrolladores ajenos al mundo académico.

Por lo tanto, la principal fuente de información se encuentra en sitios web, documentos técnicos y blogs. La investigación académica sobre blockchain ha comenzado recientemente. En 2016, Ledger, la primera revista académica dedicada a diversos aspectos (informática, ingeniería, derecho,...)

Se lanzó un programa de investigación (economía y filosofía) relacionado con la tecnología blockchain. En la comunidad de sistemas distribuidos, el enfoque se ha centrado en mejorar la seguridad o el rendimiento de los protocolos, como por ejemplo, Bitcoin-NG [Eyal et al., 2016]. En la comunidad de gestión de datos, podemos encontrar tutoriales útiles en importantes congresos, como por ejemplo, [Maiyya et al., 2018], artículos de investigación, como por ejemplo, [Dinh et al., 2018], y diseños de sistemas como BigchainDB. Comprender los cuellos de botella en el rendimiento también requiere la realización de pruebas comparativas, como se muestra en BLOCKBENCH [Dinh et al., 2018].

Ceremonias

Problema 9.1 ¿Cuál es la diferencia fundamental entre las arquitecturas P2P y cliente-servidor? ¿Es un sistema P2P con un índice centralizado equivalente a un sistema cliente-servidor? Enumere las principales ventajas y desventajas de los sistemas de intercambio de archivos P2P desde diferentes perspectivas:

- usuarios finales;
- propietarios de archivos;
- administradores de red.

Problema 9.2 ()** Una red superpuesta P2P se construye como una capa sobre una red física, generalmente Internet. Por lo tanto, tienen topologías diferentes y dos nodos vecinos en la red P2P pueden estar muy separados en la red física.

¿Cuáles son las ventajas y desventajas de esta estratificación? ¿Cuál es su impacto en el diseño de los tres tipos principales de redes P2P (no estructuradas, estructuradas y superpeer)?

Problema 9.3 (*) Considere la red P2P no estructurada de la Fig. 9.4 y el par inferior izquierdo que envía una solicitud de recurso. Ilustre y analice las dos siguientes estrategias de búsqueda en términos de completitud de resultados:

- inundaciones con TTL=3;
- cotillear con cada par que tiene una vista parcial de como máximo 3 vecinos.

Problema 9.4 (*) Considere la figura 9.7, centrada en redes estructuradas. Refine la comparación utilizando la escala del 1 al 5 (en lugar de bajo, moderado, alto) considerando los tres tipos principales de DHT: trie, hipercubo y anillo.

Problema 9.5 ()** El objetivo es diseñar una aplicación de red social P2P basada en una DHT. La aplicación debe ofrecer las funciones básicas de las redes sociales: registrar un nuevo usuario con su perfil; invitar o recuperar amigos; crear listas de amigos; publicar un mensaje a amigos; leer los mensajes de amigos; publicar un comentario en un mensaje.

Supongamos un DHT genérico con operaciones put y get, donde cada usuario es un par en el DHT.

Problema 9.6 ()** Proponer una arquitectura P2P de la aplicación de red social, con los pares (clave, datos) para las diferentes entidades que necesitan ser distribuidas.

Describa cómo se realizan las siguientes operaciones: crear o eliminar un usuario; crear o eliminar una amistad; leer mensajes de una lista de amigos. Analice las ventajas y desventajas del diseño.

Problema 9.7 (**) La misma pregunta, pero con el requisito adicional de que los datos privados (por ejemplo, el perfil del usuario) deben almacenarse en el par del usuario.

Problema 9.8 Analice las similitudes y diferencias del mapeo de esquemas en sistemas multibase de datos y sistemas P2P. En particular, compare el enfoque "local como vista" presentado en el capítulo 7 con el enfoque de mapeo de esquemas por pares de la sección 9.2.1.

Problema 9.9 (*) El algoritmo FD para el procesamiento de consultas top-k en redes P2P no estructuradas (véase el algoritmo 9.4) se basa en la inundación. Proponga una variante de FD donde, en lugar de la inundación, se utilice el método de paseo aleatorio o el cotilleo. ¿Cuáles son las ventajas y desventajas?

Problema 9.10 (*) Aplique el algoritmo TPUT (Algoritmo 9.2) a las tres listas de la base de datos de la Fig. 9.10 con $k = 3$. Para cada paso del algoritmo, muestre los resultados intermedios.

Problema 9.11 (*) La misma pregunta aplicada al algoritmo DHTop (ver algoritmo 9.5).

Problema 9.12 (*) El algoritmo 9.6 supone que las relaciones de entrada que se unirán se colocan arbitrariamente en la DHT. Suponiendo que una de las relaciones ya está codificada en los atributos de unión, proponga una mejora del algoritmo 9.6.

Problema 9.13 (*) Para mejorar la disponibilidad de datos en las DHT, una solución común consiste en replicar pares (k , datos) en varios pares mediante varias funciones hash. Esto genera el problema ilustrado en el Ejemplo 9.7. Una solución alternativa consiste en utilizar una DHT no replicada (con una sola función hash) y hacer que los nodos repliquen pares (k , datos) en algunos de sus vecinos. ¿Cuál es el efecto en el escenario del Ejemplo 9.7? ¿Cuáles son las ventajas y desventajas de este enfoque en términos de disponibilidad y equilibrio de carga?

Problema 9.14 (*) Analice las similitudes y diferencias entre las cadenas de bloques públicas y privadas (permisionadas). En particular, analice las propiedades que debe proporcionar el protocolo de validación de transacciones.

Capítulo 10

Procesamiento de Big Data



La última década ha presenciado un auge de aplicaciones "intensivas en datos" o "centradas en datos", donde el análisis de grandes volúmenes de datos heterogéneos es la base para la resolución de problemas. Estas aplicaciones se conocen comúnmente como aplicaciones de big data , y se han investigado sistemas especiales para facilitar la gestión y el procesamiento de estos datos, comúnmente conocidos como sistemas de procesamiento de big data. Estas aplicaciones surgen en diversos ámbitos, desde las ciencias de la salud hasta las redes sociales, los estudios ambientales y muchos otros. El big data es un aspecto fundamental de la ciencia de datos, que combina diversas disciplinas como la gestión de datos, el análisis y la estadística de datos, el aprendizaje automático, entre otras, para generar nuevo conocimiento a partir de ellos. Cuantos más datos haya, mejores serán los resultados de la ciencia de datos, con los consiguientes desafíos en su gestión y procesamiento.

No existe una definición precisa de aplicaciones o sistemas de big data, pero normalmente se caracterizan por las "cuatro V" (aunque también se han especificado otras, como valor, validez, etc.):

1. Volumen. Los conjuntos de datos que se utilizan en estas aplicaciones son muy grandes, típicamente del orden de los petabytes (PB; 1015 bytes) y, con el crecimiento de las aplicaciones del Internet de las Cosas, pronto alcanzarán los zettabytes (ZB; 1021 bytes). Para poner esto en perspectiva, Google informó que en 2016, las subidas de usuarios a YouTube requirieron 1 PB de nueva capacidad de almacenamiento al día. Se espera que esta cifra crezca exponencialmente, multiplicándose por diez cada cinco años (por lo que, para cuando lea este libro, su almacenamiento adicional diario podría ser de 10 PB). Facebook almacenaba alrededor de 250 000 millones de imágenes (en 2018), lo que requería exabytes de almacenamiento. Alibaba informó que, durante un período de alta actividad en 2017, se generaron 320 PB de datos de registro en un período de seis horas como resultado de la actividad de compra de los clientes.
2. Variedad. Los SGBD tradicionales (generalmente relacionales) están diseñados para funcionar con datos bien estructurados; eso es lo que describe el esquema. En las aplicaciones de big data, esto ya no es así, y es necesario gestionar y procesar datos multimodales. Además de los estructurados, los datos pueden incluir imágenes, texto, audio y vídeo. Se afirma que el 90 % de los datos generados actualmente no están estructurados.

Los sistemas de big data necesitan poder gestionar y procesar todos estos tipos de datos sin problemas.

3. Velocidad. Un aspecto importante de las aplicaciones de big data es que, en ocasiones, procesan datos que llegan al sistema a alta velocidad, lo que requiere que los sistemas puedan procesarlos conforme llegan. Siguiendo los ejemplos anteriores sobre volumen, Facebook tiene que procesar 900 millones de fotos que los usuarios suben al día; Alibaba ha informado que, durante un período pico, tuvo que procesar 470 millones de registros de eventos por segundo. Estas cifras normalmente no permiten que los sistemas almacenen los datos antes de procesarlos, lo que requiere capacidades en tiempo real.
4. Veracidad. Los datos que utilizan las aplicaciones de big data provienen de diversas fuentes, cada una de ellas puede no ser del todo fiable. Podría haber ruido, sesgo, inconsistencias entre las diferentes copias y desinformación deliberada. Esto se conoce comúnmente como "datos sucios" y es inevitable, ya que las fuentes de datos crecen junto con el volumen. Se afirma que los datos sucios cuestan más de 3000 millones de dólares al año solo en la economía estadounidense. Los sistemas de big data necesitan "limpiar" los datos y mantener su procedencia para poder determinar su fiabilidad.
Otra dimensión importante de la veracidad es la veracidad de los datos, para garantizar que no se alteren por ruido, sesgos ni manipulación intencional. El punto fundamental es que los datos deben ser fiables.

Estas características son bastante diferentes a las de los datos que manejan los SGBD tradicionales (en los que nos hemos centrado hasta ahora); requieren nuevos sistemas, metodologías y enfoques. Quizás se pueda argumentar que los SGBD paralelos (Cap. 8) gestionan el volumen razonablemente bien, dado que estos sistemas gestionan conjuntos de datos muy grandes; sin embargo, los sistemas que pueden abordar todas las dimensiones mencionadas anteriormente requieren atención. Estos son temas de investigación y desarrollo activos, y nuestro objetivo en este capítulo y en el siguiente es destacar los enfoques de infraestructura de sistemas que se están considerando actualmente para abordar los tres primeros puntos. La veracidad puede considerarse transversal a nuestra discusión y constituye un tema completo en sí mismo, por lo que no la abordaremos en mayor profundidad. En las Notas Bibliográficas, se mencionará parte de la literatura en este ámbito. Los lectores recordarán que lo abordamos brevemente en el Cap. 7 (específicamente en la Sec. 7.1.5); también abordaremos el tema en el contexto de la gestión de datos web en el Cap. 12 (específicamente en la Sec. 12.6.3).

En comparación con los SGBD tradicionales, la gestión de big data utiliza una pila de software diferente con las siguientes capas (véase la Fig. 10.1). La gestión de big data se basa en una capa de almacenamiento distribuido, donde los datos se almacenan típicamente en archivos u objetos distribuidos en los nodos de un clúster sin recursos compartidos. Un marco de procesamiento de datos accede directamente a los datos almacenados en archivos distribuidos, lo que permite a los programadores expresar código de procesamiento paralelo sin la intervención de un SGBD. Además de los marcos de procesamiento de datos, podrían existir herramientas de scripting y consultas declarativas (similares a SQL). Para la gestión de datos multimodales, se suelen implementar sistemas NoSQL como parte de la capa de acceso a datos, o bien, se puede utilizar un motor de streaming, o incluso motores de búsqueda. Finalmente, en la capa superior se proporcionan diversas herramientas que permiten desarrollar análisis de big data más complejos, incluyendo herramientas de aprendizaje automático (ML). Esta pila de software, como se ejemplifica con Hadoop, que analizare-

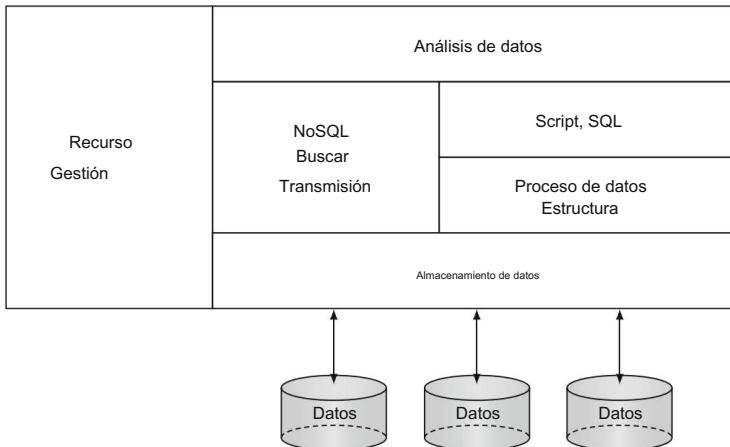


Fig. 10.1 Pila de software de gestión de big data

Fomenta la integración de componentes débilmente acoplados (normalmente de código abierto). Por ejemplo, un SGBD NoSQL suele ser compatible con diferentes sistemas de almacenamiento (p. ej., HDFS, etc.). Estos sistemas se suelen implementar en entornos de computación en la nube, ya sean públicos o privados. Esta arquitectura de pila de software guiará nuestro análisis en este capítulo y en el siguiente.

El resto de este capítulo se centra en los componentes de esta arquitectura de abajo a arriba y, en el proceso, abordamos dos de las V que caracterizan a los sistemas de big data. La sección [\(10.1\)](#) se centra en los sistemas de almacenamiento distribuido. La sección [10.2](#) cubre dos importantes marcos de procesamiento de big data, centrándose en MapReduce y Spark. Junto con la sección [10.1](#), esta sección aborda las preocupaciones de escalabilidad, es decir, la dimensión de "volumen". En la sección [10.3](#) analizamos el procesamiento de datos para datos de flujo; esto aborda la dimensión de "velocidad". En la sección [10.4](#) cubrimos los sistemas de grafos enfocándonos en el análisis de grafos, abordando algunos de los problemas de "variedad". Los problemas de variedad también se abordan en la sección [10.5](#) donde analizamos el campo emergente de los lagos de datos. Los lagos de datos integran datos de muchas fuentes que pueden o no estar estructurados. Dejamos el lado NoSQL de esta arquitectura para el siguiente capítulo (cap. 11).

10.1 Sistemas de almacenamiento distribuido

La gestión de big data se basa en una capa de almacenamiento distribuido, donde los datos se almacenan generalmente en archivos u objetos distribuidos entre los nodos de un clúster sin recursos compartidos. Esta es una diferencia importante con la pila de software de los SGBD actuales, que se basa en el almacenamiento en bloques. La historia de los SGBD es interesante para comprender la evolución de esta pila de software. Los primeros SGBD, basados en modelos jerárquicos o de red, se crearon como extensiones de un sistema de archivos, como COBOL, con interconexión de a

Los enlaces, y también los primeros SGBD relacionales, se construyeron sobre un sistema de archivos. Por ejemplo, el famoso SGBD INGRES se implementó sobre el sistema de archivos Unix. Pero el uso de un sistema de archivos de propósito general estaba haciendo que el acceso a los datos fuera bastante ineficiente, ya que el DBMS no podía tener control sobre la agrupación de datos en disco ni sobre la gestión de caché en la memoria principal. La principal crítica a este enfoque basado en archivos era la falta de compatibilidad del sistema operativo para la gestión de bases de datos (en aquel momento). Como resultado, la arquitectura de los DBMS relacionales evolucionó de basada en archivos a basada en bloques, utilizando una interfaz de disco sin procesar proporcionada por el sistema operativo. Una interfaz basada en bloques proporciona acceso directo y eficiente a los bloques de disco (la unidad de asignación de almacenamiento en discos). Hoy en día, todos los DBMS relacionales están basados en bloques y, por lo tanto, tienen control total sobre la gestión de discos. La evolución hacia los DBMS paralelos mantuvo el mismo enfoque, principalmente para facilitar la transición desde sistemas centralizados. Una razón principal para el retorno al uso de un sistema de archivos es que el almacenamiento distribuido puede hacerse tolerante a fallos y escalable, lo que facilita la construcción de las capas superiores de gestión de datos.

En este contexto, la capa de almacenamiento distribuido suele ofrecer dos soluciones para almacenar datos, objetos o archivos distribuidos en los nodos del clúster. Estas dos soluciones son complementarias, ya que tienen diferentes propósitos y pueden combinarse.

El almacenamiento de objetos gestiona los datos como objetos. Un objeto incluye sus datos junto con una cantidad variable de metadatos y un identificador único (oid) en un espacio de objetos plano. Por lo tanto, un objeto puede representarse como un triple oid: datos y metadatos. Una vez creado, se puede acceder a él directamente mediante su oid. El hecho de que los datos y los metadatos estén agrupados dentro de cada objeto facilita su traslado entre ubicaciones distribuidas. A diferencia de los sistemas de archivos, donde el tipo de metadatos es el mismo para todos los archivos, los objetos pueden tener cantidades variables de metadatos. Esto ofrece al usuario una gran flexibilidad para definir cómo se protegen los objetos, cómo se pueden replicar, cuándo se pueden eliminar, etc. El uso de un espacio de objetos plano permite gestionar cantidades masivas (por ejemplo, miles de millones o billones) de objetos de datos no estructurados. Finalmente, se puede acceder fácilmente a los objetos mediante una API sencilla basada en REST con comandos put y get fáciles de usar en protocolos de Internet. Los almacenes de objetos son especialmente útiles para almacenar una gran cantidad de objetos de datos relativamente pequeños, como fotos, archivos adjuntos de correo, etc. Por lo tanto, este enfoque ha sido popular entre la mayoría de los proveedores de nube que prestan servicios a estas aplicaciones.

El almacenamiento de archivos administra datos dentro de archivos no estructurados (es decir, secuencias de bytes) sobre los cuales los datos pueden organizarse como registros de longitud fija o variable. Un sistema de archivos organiza los archivos en una jerarquía de directorios y mantiene para cada archivo sus metadatos (nombre de archivo, posición en la carpeta, propietario, longitud del contenido, hora de creación, hora de la última actualización, permisos de acceso, etc.), separados del contenido del archivo. Por lo tanto, primero se deben leer los metadatos del archivo para localizar el contenido del archivo. Debido a esta gestión de metadatos, el almacenamiento de archivos es apropiado para compartir archivos localmente dentro de un centro de datos y cuando el número de archivos es limitado (por ejemplo, en cientos de miles). Para gestionar archivos grandes que pueden contener una gran cantidad de registros, los archivos deben dividirse y distribuirse en varios nodos del clúster mediante un sistema de archivos distribuido. Uno de los sistemas de archivos distribuidos más influyentes es Google File System (GFS). En el resto de esta sección, describiremos GFS. También analizaremos la combinación de almacenamiento de objetos y almacenamiento de archivos, que suele ser útil en la nube.

10.1.1 Sistema de archivos de Google

GFS ha sido desarrollado por Google para su uso interno y lo utilizan muchas aplicaciones y sistemas de Google, como Bigtable.

Al igual que otros sistemas de archivos distribuidos, GFS busca proporcionar rendimiento, escalabilidad, tolerancia a fallos y disponibilidad. Sin embargo, los sistemas objetivo, los clústeres sin recursos compartidos, presentan un desafío, ya que están compuestos por muchos (por ejemplo, miles) servidores construidos con hardware económico. Por lo tanto, la probabilidad de que un servidor falle en un momento dado es alta, lo que dificulta la tolerancia a fallos. GFS aborda este problema mediante la replicación y la conmutación por error, como se explica más adelante. También está optimizado para aplicaciones de Google con uso intensivo de datos, como motores de búsqueda o análisis de datos.

Estas aplicaciones tienen las siguientes características: en primer lugar, sus archivos son muy grandes, generalmente de varios gigabytes, y contienen numerosos objetos, como documentos web.

En segundo lugar, las cargas de trabajo consisten principalmente en operaciones de lectura y anexión, mientras que las actualizaciones aleatorias son poco frecuentes. Las operaciones de lectura consisten en grandes lecturas de datos masivos (p. ej., 1 MB) y pequeñas lecturas aleatorias (p. ej., unos pocos KB). Las operaciones de anexión también son grandes, y puede haber muchos clientes simultáneos que anexen el mismo archivo. En tercer lugar, dado que las cargas de trabajo consisten principalmente en grandes operaciones de lectura y anexión, un alto rendimiento es más importante que una baja latencia.

GFS organiza los archivos como un trie de directorios y los identifica por sus rutas de acceso. Proporciona una interfaz de sistema de archivos con operaciones de archivo tradicionales (crear, abrir, leer, escribir, cerrar y eliminar archivos) y dos operaciones adicionales: instantánea, que permite crear una copia de un archivo o de un trie de directorios, y anexión de registros, que permite que clientes concurrentes anexen datos (el "registro") a un archivo de forma eficiente.

Un registro se añade de forma atómica, es decir, como una cadena de bytes continua, en una ubicación de byte determinada por GFS. Esto evita la necesidad de la gestión de bloqueos distribuidos que sería necesaria con la operación de escritura tradicional (que podría utilizarse para añadir datos).

La arquitectura de GFS se ilustra en la Fig. 10.2. Los archivos se dividen en particiones de tamaño fijo, llamadas fragmentos, de gran tamaño (64 MB). Los nodos del clúster constan de clientes GFS que proporcionan la interfaz GFS a las aplicaciones, servidores de fragmentos que almacenan fragmentos y un único maestro GFS que mantiene los metadatos de los archivos, como el espacio de nombres.

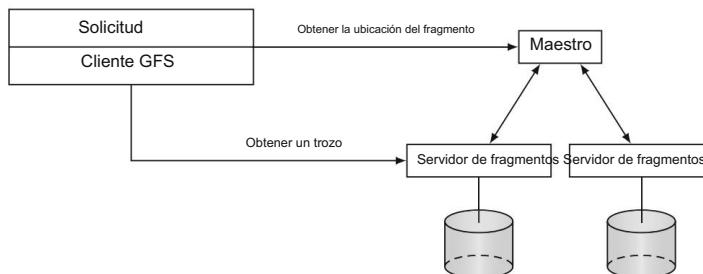


Figura 10.2 Arquitectura GFS

Información de control de acceso e información de ubicación de los fragmentos. Cada fragmento tiene un ID único asignado por el maestro al momento de su creación y, por razones de confiabilidad, se replica en al menos tres servidores de fragmentos. Para acceder a los datos de los fragmentos, un cliente primero debe solicitar al maestro la ubicación de los fragmentos, necesaria para responder al acceso a los archivos de la aplicación. Luego, utilizando la información devuelta por el maestro, el cliente puede solicitar los datos del fragmento a una de las réplicas.

Esta arquitectura que utiliza un solo maestro es simple y, dado que el maestro se utiliza principalmente para ubicar fragmentos y no contiene datos de fragmentos, no es un cuello de botella.

Además, no hay almacenamiento en caché de datos ni en los clientes ni en los servidores de fragmentos, ya que no beneficiaría las lecturas de gran volumen. Otra simplificación es un modelo de consistencia relajada para escrituras concurrentes y anexiones de registros. Por lo tanto, las aplicaciones deben gestionar la consistencia relajada mediante técnicas como la creación de puntos de control y la escritura de registros autovalidados. Finalmente, para mantener la alta disponibilidad del sistema ante fallos frecuentes de nodos, GFS se basa en la replicación y la comutación por error automática. Cada fragmento se replica en varios servidores (por defecto, GFS almacena tres réplicas). El servidor maestro envía periódicamente mensajes de latido a cada servidor de fragmentos. Luego, ante un fallo de un servidor de fragmentos, el servidor maestro realiza una comutación por error automática, redirigiendo todos los accesos a los archivos a un servidor activo que contiene una réplica. GFS también replica todos los datos del servidor maestro a un servidor maestro en la sombra, de modo que, en caso de fallo de un servidor maestro, este toma el control automáticamente.

Existen implementaciones de código abierto de GFS, como el Sistema de Archivos Distribuidos Hadoop (HDFS), que se analiza en la Sección 10.2.1. Existen otros sistemas de archivos distribuidos de código abierto importantes para sistemas de clúster, como GlusterFS para sistemas sin recursos compartidos y el Sistema de Archivos Global 2 (GFS2) para discos compartidos, ambos desarrollados por Red Hat para Linux.

10.1.2 Combinación de almacenamiento de objetos y almacenamiento de archivos

Una tendencia importante es combinar el almacenamiento de objetos y archivos en un único sistema para admitir tanto grandes cantidades de objetos como archivos de gran tamaño. El primer sistema que combinó el almacenamiento de objetos y archivos fue Ceph. Ceph es una plataforma de almacenamiento de software de código abierto, ahora desarrollada por Red Hat en un clúster sin recursos compartidos a escala de exabytes. Ceph desacopla las operaciones de datos y metadatos eliminando las tablas de asignación de archivos y sustituyéndolas por funciones de distribución de datos diseñadas para clústeres heterogéneos y dinámicos de dispositivos de almacenamiento de objetos (OSD) poco fiables. Esto permite a Ceph aprovechar la inteligencia presente en los OSD para distribuir la complejidad del acceso a los datos, la serialización de actualizaciones, la replicación y la fiabilidad, la detección de fallos y la recuperación. Ceph y GlusterFS son ahora las dos principales plataformas de almacenamiento que ofrece Red Hat para clústeres sin recursos compartidos.

HDFS, por otro lado, se ha convertido en el estándar de facto para la gestión escalable y confiable de sistemas de archivos para big data. Por lo tanto, existe un gran incentivo para agregar capacidades de almacenamiento de objetos a HDFS, con el fin de facilitar el almacenamiento de datos a los proveedores y usuarios de la nube. En Azure HDInsight, la solución de Microsoft basada en Hadoop para...

HDFS, que gestiona big data en la nube, se integra con Azure Blob Storage, el administrador de almacenamiento de objetos, para operar directamente con datos estructurados o no estructurados. Los contenedores de Blob Storage almacenan datos como pares clave-valor, sin jerarquía de directorios.

10.2 Marcos de procesamiento de big data

Un importante tipo de aplicaciones de big data requiere la gestión de datos sin la sobrecarga que supone la gestión completa de bases de datos, y los servicios en la nube requieren escalabilidad para aplicaciones que se puedan dividir fácilmente en varias tareas paralelas, pero más pequeñas: las llamadas aplicaciones paralelizables. Para estos casos, donde la escalabilidad es más importante que las consultas declarativas, la compatibilidad con transacciones y la consistencia de la base de datos, se ha propuesto una plataforma de procesamiento paralelo llamada MapReduce. La idea fundamental es simplificar el procesamiento paralelo mediante una plataforma de computación distribuida que ofrece únicamente dos interfaces: map y reduce. Los programadores implementan sus propias funciones de map y reduce , mientras que el sistema se encarga de programar y sincronizar dichas tareas. Esta arquitectura se optimiza aún más en Spark, por lo que gran parte de la siguiente explicación se aplica a ambos frameworks.

Comenzamos discutiendo el MapReduce básico (Sección 10.2.1) y luego presentamos las optimizaciones de Spark (Sección 10.2.2).

Las ventajas comúnmente citadas de este tipo de marco de procesamiento son las siguientes:

1. Flexibilidad. Dado que el código de las funciones map y reduce lo escribe el usuario, existe una considerable flexibilidad para especificar el procesamiento exacto requerido sobre los datos, en lugar de especificarlo mediante SQL. Los programadores pueden escribir funciones map y reduce sencillas para procesar grandes volúmenes de datos en varias máquinas (o nodos, como se suele usar en los SGBD paralelos)¹ sin necesidad de saber cómo parallelizar el procesamiento de una tarea MapReduce.
2. Escalabilidad. Un desafío importante en muchas aplicaciones existentes es la capacidad de escalar con el aumento del volumen de datos. En particular, en las aplicaciones en la nube se busca una escalabilidad elástica , lo que requiere que el sistema pueda ajustar su rendimiento dinámicamente según cambien los requisitos de computación.
Este modelo de servicio de “pago por uso” ha sido ampliamente adoptado por los proveedores de servicios de computación en la nube, y MapReduce puede soportarlo sin problemas mediante la ejecución paralela de datos.
3. Eficiencia. MapReduce no necesita cargar datos en una base de datos, lo que evita el alto coste de la ingesta de datos. Por lo tanto, es muy eficiente para aplicaciones que requieren procesar los datos solo una vez (o pocas veces).

En la literatura de MapReduce, estos se conocen comúnmente como trabajadores, mientras que en el capítulo sobre SGBD paralelos y el capítulo posterior sobre NoSQL, utilizamos el término nodo . El lector debe tener en cuenta que usamos estos términos indistintamente.

4. Tolerancia a fallos. En MapReduce, cada trabajo se divide en varias tareas pequeñas que se asignan a diferentes máquinas. El fallo de una tarea o máquina se compensa asignándola a una máquina capaz de gestionar la carga. La entrada de un trabajo se almacena en un sistema de archivos distribuido donde se mantienen múltiples réplicas para garantizar una alta disponibilidad. Por lo tanto, la tarea de mapeo fallida puede repetirse correctamente recargando la réplica. La tarea de reducción fallida también puede repetirse volviendo a extraer los datos de las tareas de mapeo completadas.

Las críticas a MapReduce se centran en su funcionalidad reducida, que requiere un esfuerzo de programación considerable, y su inadecuación para ciertos tipos de aplicaciones (por ejemplo, aquellas que requieren cálculos iterativos). MapReduce no requiere la existencia de un esquema ni proporciona un lenguaje de alto nivel como SQL. La ventaja de flexibilidad mencionada anteriormente se consigue a costa de una programación considerable (y generalmente sofisticada) por parte del usuario.

Por consiguiente, un trabajo que se puede realizar con comandos SQL relativamente simples puede requerir una cantidad considerable de programación en MapReduce, y este código generalmente no es reutilizable. Además, MapReduce no cuenta con soporte integrado para indexación ni optimización de consultas, por lo que siempre recurre a escaneos (esto se destaca tanto como una ventaja como una desventaja según el punto de vista).

10.2.1 Procesamiento de datos de MapReduce

Como se mencionó anteriormente, MapReduce es un enfoque simplificado de procesamiento de datos paralelos para su ejecución en un clúster de computadoras. Permite a los programadores expresar de forma sencilla y funcional sus cálculos en grandes conjuntos de datos y oculta los detalles del procesamiento de datos paralelos, el balanceo de carga y la tolerancia a fallos. Su modelo de programación consta de dos funciones definidas por el usuario, `map()` y `reduce()`, con la siguiente semántica:

mapa	$(k1, v1) \rightarrow \text{lista } (k2, v2)$
reducir	$(k2, \text{lista } (v2)) \rightarrow \text{lista } (v3)$

La función `map` se aplica a cada registro del conjunto de datos de entrada para calcular cero o más pares intermedios (clave-valor). La función `reduce` se aplica a todos los valores que comparten la misma clave única para calcular un resultado combinado. Dado que funcionan con entradas independientes, `map` y `reduce` pueden procesarse automáticamente en paralelo en diferentes particiones de datos utilizando varios nodos del clúster.

La Figura 10.3 ofrece una descripción general de la ejecución de MapReduce en un clúster. Las entradas de la función de mapa son un conjunto de pares clave-valor. Cuando se envía un trabajo de MapReduce al sistema, las tareas de mapa (procesos denominados mapeadores) se inicien en los nodos de cómputo y cada tarea de mapa aplica la función de mapa a cada par clave-valor ($k1, v1$) que le está asignado. Cero o más intermedios

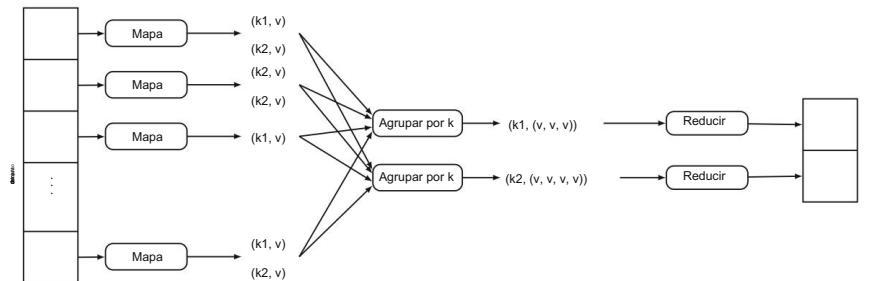


Fig. 10.3 Descripción general de la ejecución de MapReduce

Se pueden generar pares clave/valor (lista (k2, v2)) para el mismo par clave/valor de entrada. Estos resultados intermedios se almacenan en el sistema de archivos local y se ordenan por las claves. Una vez completadas todas las tareas del mapa, el motor MapReduce notifica a las tareas de reducción (que también son procesos denominados reductores) para que comiencen su procesamiento. Los reductores extraerán los archivos de salida de las tareas de mapa en paralelo y ordenarán mediante fusión los archivos obtenidos para combinar los pares clave-valor en un nuevo conjunto de pares clave-valor (k2, lista (v2)). Todos los valores con la misma clave k2 se agrupan en una lista y se utilizan como entrada para la función de reducción (comúnmente conocida como proceso de reorganización , que consiste en una ordenación paralela). La función de reducción aplica la lógica de procesamiento definida por el usuario para procesar los datos. Los resultados, normalmente una lista de valores, se vuelven a escribir en el sistema de almacenamiento.

Además de escribir el mapa y reducir las funciones, los programadores pueden ejercer un mayor control (por ejemplo, formatos de entrada/salida y función de partición) mediante funciones definidas por el usuario (UDF) que proporcionan estos sistemas.

Ejemplo 10.1 Consideremos la relación EMP(ENO, ENAME, TITLE, CITY) y la siguiente consulta SQL que devuelve para cada ciudad, el número de empleados cuyo nombre contiene "Smith".

SELECCIONAR CIUDAD, CUENTA(*)
DESDE EMP
DONDE ENAME COMO "%Smith"
GRUPO POR CIUDAD

El procesamiento de esta consulta con MapReduce se puede realizar con el siguiente mapa y Reducir funciones (que damos en pseudocódigo).

Mapa (Entrada: (TID,EMP), Salida: (CIUDAD,1))
 si EMP.ENAME como "%Smith" devuelve (CIUDAD,1)
Reducir (Entrada: (CIUDAD,lista(1)), Salida: (CIUDAD, SUMA(lista(1)))
 devolver (CIUDAD,SUMA(1))

El mapa se aplica en paralelo a cada tupla de EMP. Toma un par (TID, EMP), donde la clave es el identificador de tupla de EMP (TID) y el valor es la tupla de EMP.

Si corresponde, devuelve un par (CIUDAD,1). Tenga en cuenta que el análisis del formato de tupla para extraer atributos debe realizarse mediante la función map . A continuación, se agrupan todos los pares (CIUDAD,1) con la misma CIUDAD y se crea un par (CIUDAD,lista(1)) para cada CIUDAD. A continuación, se aplica la función reduce en paralelo para calcular el recuento de cada CIUDAD y generar el resultado de la consulta.

10.2.1.1 Arquitectura de MapReduce

Al analizar los aspectos específicos de MapReduce, nos centraremos en una implementación específica: Hadoop. La pila de Hadoop se muestra en la Fig. 10.4, que es una implementación específica de la arquitectura de big data descrita en la Fig. 10.1. Hadoop utiliza el Sistema de Archivos Distribuidos de Hadoop (HDFS) como almacenamiento, aunque puede implementarse en diferentes sistemas de almacenamiento. HDFS y el motor de procesamiento de Hadoop están débilmente conectados; pueden compartir el mismo conjunto de nodos de cómputo o implementarse en nodos diferentes. En HDFS, se crean dos tipos de nodos: el nodo de nombre y el nodo de datos. El nodo de nombre registra cómo se partitionan los datos y supervisa el estado de los nodos de datos en HDFS. Los datos importados a HDFS se dividen en fragmentos de igual tamaño y el nodo de nombre distribuye los fragmentos de datos a diferentes nodos de datos que almacenan y gestionan los fragmentos asignados. El nodo de nombre también actúa como servidor de diccionario, proporcionando información de particionamiento a las aplicaciones que buscan un fragmento específico de datos.

La disociación del motor de procesamiento de Hadoop del sistema de almacenamiento subyacente permite que las capas de procesamiento y almacenamiento escalen vertical y horizontalmente de forma independiente según sea necesario. En la sección 10.1, analizamos diferentes enfoques para el diseño de sistemas de almacenamiento distribuido y proporcionamos ejemplos. Cada fragmento de datos almacenado en cada máquina del clúster constituye una entrada para un mapeador. Por lo tanto, si el conjunto de datos se partitiona en k fragmentos, Hadoop creará k mapeadores para procesar los datos (o viceversa).

El motor de procesamiento de Hadoop tiene dos tipos de nodos: el nodo maestro y los nodos de trabajo , como se muestra en la figura 10.5. El nodo maestro controla el flujo de ejecución.

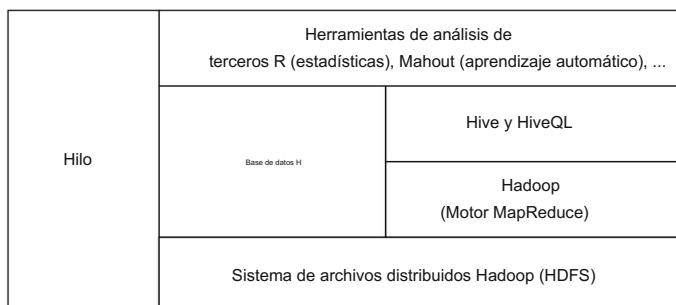


Figura 10.4 Pila de Hadoop

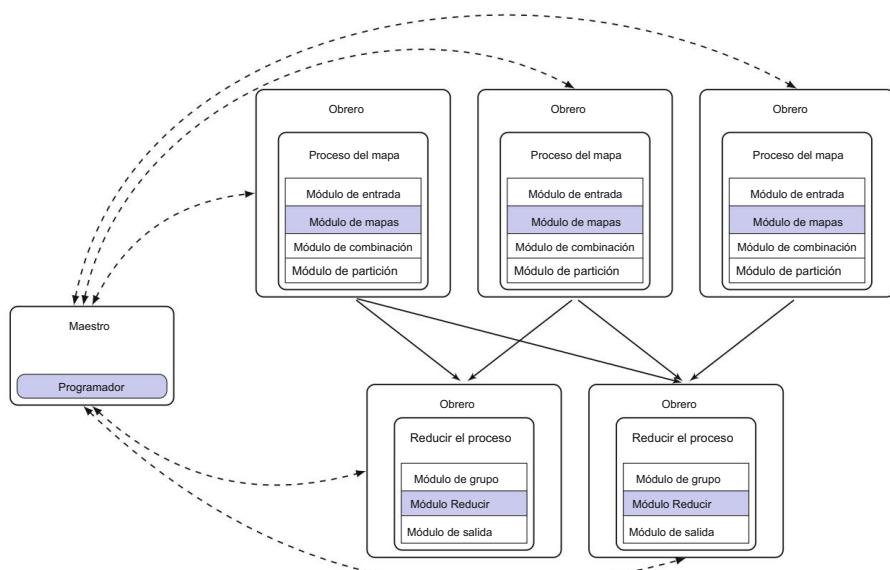


Fig. 10.5 Arquitectura maestro-trabajador de MapReduce

Las tareas en los nodos de trabajo se gestionan mediante el módulo planificador (en Hadoop, se conoce como rastreador de tareas). Cada nodo de trabajo es responsable de una tarea de mapeo o reducción. La implementación básica del motor MapReduce debe incluir los siguientes módulos, los tres primeros esenciales; los restantes son extensiones:

1. Planificador. El planificador se encarga de asignar las tareas de mapeo y reducción a los nodos de trabajo según la ubicación de los datos, el estado de la red y otras estadísticas de los nodos. También controla la tolerancia a fallos reprogramando un proceso fallido a otros nodos (si es posible). El diseño del planificador afecta significativamente el rendimiento del sistema MapReduce.
2. Módulo de mapa. El módulo de mapa escanea un fragmento de datos e invoca la función `map()` definida por el usuario para procesar los datos de entrada. Tras generar los resultados intermedios (un conjunto de pares clave-valor), los agrupa según las claves de partición, ordena las tuplas en cada partición y notifica al nodo maestro la posición de los resultados.
3. Módulo de reducción. El módulo de reducción extrae datos de los mapeadores tras recibir la notificación del maestro. Una vez obtenidos todos los resultados intermedios de los mapeadores, el reductor fusiona los datos por claves y agrupa todos los valores con la misma clave. Finalmente, la función definida por el usuario se aplica a cada par clave-valor y los resultados se exportan al almacenamiento distribuido.
4. Módulos de entrada y salida. El módulo de entrada se encarga de reconocer los datos de entrada con diferentes formatos y de dividirlos en pares clave-valor. Este módulo permite que el motor de procesamiento funcione con diferentes sistemas de almacenamiento, permitiendo el uso de diferentes formatos de entrada para analizar distintos datos.

Fuentes, como archivos de texto, archivos binarios e incluso archivos de bases de datos. El módulo de salida también especifica el formato de salida de los mapeadores y reductores.

5. Módulo de combinación. El propósito de este módulo es reducir el coste de la reorganización mediante un proceso de reducción local para los pares clave-valor generados por el asignador.
6. Módulo de partición. Se utiliza para especificar cómo se redistribuyen los pares clave-valor de los mapeadores a los reductores. La función de partición predeterminada se define como $f(key) = h(key) \%numOfReducer$, donde $\%$ indica el operador mod y $h(key)$ es el valor hash de la clave. Se envía un par clave-valor (k, v) al reductor $f(k)$. Los usuarios pueden definir diferentes funciones de partición para admitir un comportamiento más sofisticado.
7. Módulo de grupo. Este módulo especifica cómo combinar los datos recibidos de diferentes procesos de mapa en una sola ejecución ordenada durante la fase de reducción. Al especificar la función de grupo, que es una función de la clave de salida del mapa, los datos se pueden combinar con mayor flexibilidad. Por ejemplo, si la clave de salida del mapa está compuesta por varios atributos (IP de origen, URL de destino), la función de grupo solo puede comparar un subconjunto de los atributos (IP de origen). Como resultado, en el módulo reductor, la función de reducción se aplica a los pares clave-valor con la misma IP de origen.

Dado su propósito declarado de escalar sobre un gran número de nodos de procesamiento, un sistema MapReduce debe ser eficiente en tolerancia a fallos. Cuando una tarea de mapeo o reducción falla, se crea otra tarea en una máquina diferente para reejecutar la tarea fallida. Dado que el mapeador almacena los resultados localmente, incluso una tarea de mapeo completada debe reejecutarse en caso de un fallo de nodo. Por el contrario, dado que el reductor almacena los resultados en el almacenamiento distribuido, una tarea de reducción completada no necesita reejecutarse cuando se produce un fallo de nodo.

10.2.1.2 Lenguajes de alto nivel para MapReduce

La filosofía de diseño de MapReduce es proporcionar un marco flexible que pueda aprovecharse para resolver diferentes problemas. Por lo tanto, MapReduce no proporciona un lenguaje de consulta, esperando que los usuarios implementen sus funciones `map()` y `reduce()` personalizadas . Si bien esto proporciona una flexibilidad considerable, añade complejidad al desarrollo de aplicaciones. Para facilitar el uso de MapReduce, se han desarrollado varios lenguajes de alto nivel, algunos de los cuales son declarativos (HiveQL, Tenzing, JAQL), otros son lenguajes de flujo de datos (Pig Latin), lenguajes procedimentales (Sawzall), bibliotecas Java (FlumeJava) y otros son lenguajes declarativos de aprendizaje automático (SystemML). Desde la perspectiva de un sistema de bases de datos, quizás los lenguajes declarativos sean de mayor interés. Aunque estos lenguajes son diferentes, generalmente siguen una arquitectura similar, como se muestra en la Fig. 10.6. El nivel superior consta de múltiples interfaces de consulta, como la interfaz de línea de comandos, la interfaz web o el servidor JDBC/ODBC. Actualmente, solo Hive admite todas estas interfaces de consulta. Tras emitir una consulta desde una de las interfaces, el compilador de consultas la analiza para generar un plan lógico utilizando los metadatos. A continuación, la regla...

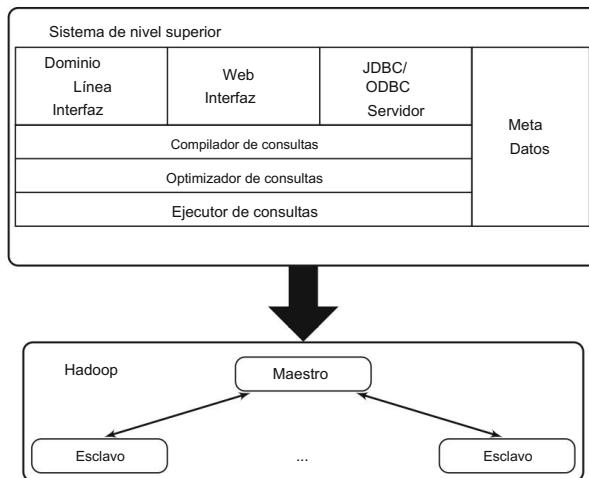


Figura 10.6 Arquitectura de las implementaciones de consultas declarativas

Se aplica una optimización basada en datos, como la reducción de la proyección, para optimizar el plan lógico. Finalmente, el plan se transforma en un grafo acíclico dirigido (DAG) de trabajos de MapReduce, que posteriormente se envían al motor de ejecución uno por uno.

10.2.1.3 Implementación de operadores de base de datos en MapReduce

Si se utilizan implementaciones de MapReduce como Hadoop para la gestión de datos que van más allá de las aplicaciones "extremadamente paralelizables", es importante implementar operadores de bases de datos típicos en estos sistemas, lo cual ha sido objeto de investigación. Operadores simples como select y project pueden integrarse fácilmente en la función map , mientras que los complejos, como theta-join, equijoin y multiway join, requieren un esfuerzo considerable. En esta sección, analizamos estas implementaciones.

La proyección y la selección se pueden implementar fácilmente agregando algunas condiciones a la función de mapa para filtrar las columnas y tuplas innecesarias.

La agregación se puede implementar fácilmente mediante las funciones map() y reduce() ; la figura 10.7 ilustra el flujo de datos del trabajo MapReduce para el operador de agregación. El mapeador extrae una clave de agregación (Aid) para cada tupla entrante (transformada en un par clave-valor). Las tuplas con la misma clave de agregación se redistribuyen a los mismos reductores, y la función de agregación (p. ej., suma, min) se aplica a estas tuplas.

Las implementaciones del operador de unión han atraído la mayor atención, ya que es uno de los operadores más costosos y una mejor implementación puede conducir potencialmente a

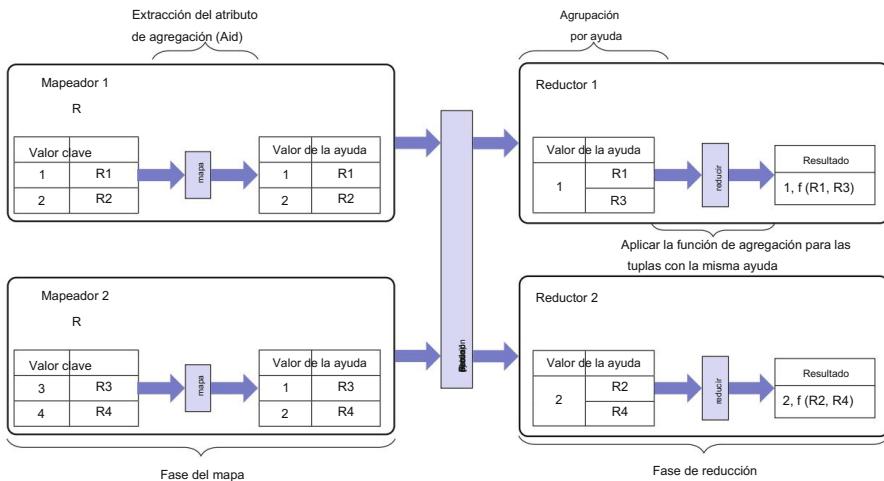


Fig. 10.7 Flujo de datos de agregación

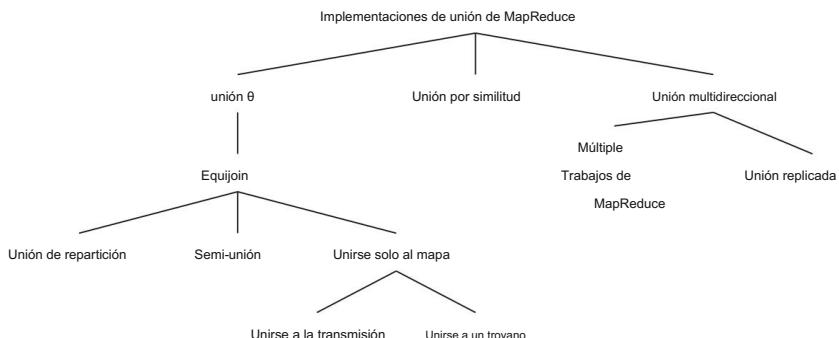


Fig. 10.8 Implementaciones de unión en MapReduce

Mejora significativa del rendimiento. Los algoritmos de unión existentes se resumen en la figura 10.8. Describiremos implementaciones de unión theta y equijoin como ejemplos.

Recuerde que theta-join (θ -join) es un operador de unión donde la condición de unión θ es una de $\{<, \leq, =, \geq, >, \neq\}$. Una unión binaria (natural) de las relaciones $R(A, B)$ y $S(B, C)$ se puede realizar usando MapReduce de la siguiente manera. La relación R se partitiona y cada partición se asigna a un conjunto de mapeadores. Cada mapeador toma las tuplas a, b y las convierte en una lista de pares clave/valor de la forma (b, a, R) , donde la clave es el atributo de unión y el valor incluye el nombre de la relación R . Estos pares clave/valor se barajan y se envían a los reductores para que todos los pares con el mismo valor de clave de unión se recopilen en el mismo reductor. El mismo proceso se aplica a S . Cada reductor luego une tuplas de R con tuplas de S (la inclusión del nombre de la relación en el valor garantiza que las tuplas de R o S no se unan entre sí).

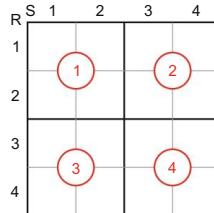


Figura 10.9 Mapeo de matriz a reductor para producto vectorial

Para implementar eficientemente la unión theta en MapReduce, las tuplas $|R| \times |S|$ deben distribuirse uniformemente en los reductores R, de modo que cada reductor genere aproximadamente el mismo número de resultados: el algoritmo $|R| \times |S|$ 1-Bucket-Theta logra esto dividiendo uniformemente la matriz de unión en cubos (Fig. 10.9) y asignando cada cubo a un solo reductor para eliminar la duplicación de cálculos. Este algoritmo, a su vez, garantiza que todos los reductores tengan asignado el mismo número de cubos para equilibrar la carga. En la Fig. 10.9, las tablas R y S están divididas uniformemente en 4 partes, lo que resulta en una matriz con 16 cubos agrupados en 4 regiones. Cada región está asignada a un reductor.

La Figura 10.10 ilustra el flujo de datos de la unión theta cuando θ es igual a “=” para el caso mostrado en la Figura 10.9. Las fases de mapeo y reducción se implementan de la siguiente manera:

1. Mapa. En el lado del mapa, para cada tupla de R o S, se selecciona aleatoriamente un identificador de fila o columna (llamado Bid) entre 1 y el número de regiones (4 en el ejemplo anterior) como clave de salida del mapa. La tupla se concatena con una etiqueta que indica su origen como valor de salida del mapa. El Bid especifica a qué fila o columna de la matriz (Fig. 10.9) pertenece la tupla, y las tuplas de salida de la función map() se redistribuyen entre todos los reductores (cada reductor corresponde a una región) que intersecan con la fila o columna.
2. Reducir. En la fase de reducción, las tuplas de la misma tabla se agrupan según las etiquetas. El cálculo de la unión theta local se aplica a las dos particiones. Los resultados calificados ($R.key = S.key$) se envían al almacenamiento. Dado que cada contenedor se asigna a un solo reductor, no se generan resultados redundantes.

En la Fig. 10.9 hay 16 contenedores organizados en 4 regiones; en la Fig. 10.10 hay 4 reductores, cada uno responsable de una región. Dado que el Reductor 1 está a cargo de la región 1, todas las R tuplas con Bid = 1 o 2 y las S tuplas con Bid = 1 o 2 se le envían. De forma similar, el Reductor 2 recibe R tuplas con Bid = 1 o 2 y las S tuplas con Bid = 3 o 4. Cada reductor divide las tuplas que recibe en dos partes según sus orígenes y las une.

Consideremos ahora equijoin, que es un caso especial de θ -join donde θ es “=”.

Existen tres variantes de la implementación de la unión equitativa: unión por repartición, unión basada en semiunión y unión solo por mapa. A continuación, se detalla la unión por repartición. La implementación basada en Semijoin consta de tres trabajos de MapReduce: el primero es un trabajo completo

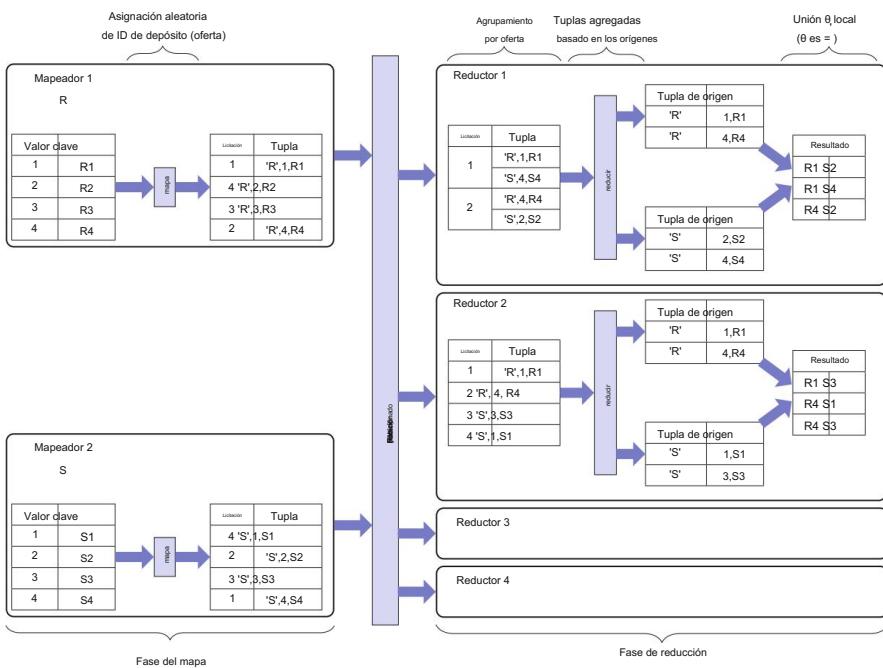


Fig. 10.10 Flujo de datos de unión theta (theta es igual a "=")

Trabajo de MapReduce que extrae las claves de unión únicas de una de las relaciones, digamos R, donde la tarea de mapa extrae la clave de unión de cada tupla y mezcla las claves idénticas al mismo reducer, y la tarea de reducción elimina las claves duplicadas y las almacena. Los resultados en DFS como un conjunto de archivos (u_0, u_1, \dots, u_k). El segundo trabajo es solo de mapa. trabajo que produce los resultados de semiunión $S = S R$. En este trabajo, dado que los archivos que Las claves únicas de R son pequeñas y se transmiten a cada asignador y localmente. Se unió a la parte de S (denominada fragmento de datos) asignada a ese asignador. El tercer trabajo También es un trabajo exclusivo de mapas donde S se transmite a todos los mapeadores y se une localmente con R. La unión solo de mapa requiere solo procesamiento del lado del mapa. Si la relación interna es mucho más pequeña que la relación externa, entonces se puede evitar la mezcla (como se propone) en la unión de difusión) mediante una tarea de mapa similar al tercer trabajo basado en semiunión algoritmo. Suponiendo que S es la relación interna y R es la externa, cada mapeador carga la tabla S completa para construir un hash en memoria y escanea su fragmento de datos asignado de R (es decir, R_i). La unión hash local se realiza entre S y R_i .

La unión de repartición es el algoritmo de unión predeterminado para MapReduce en Hadoop. Se dividen dos tablas en la fase de mapa, seguida de la mezcla de las tuplas con La misma clave para el mismo reducer que une las tuplas. Como se muestra en la Fig. 10.11, La unión de repartición se puede implementar como un trabajo de MapReduce.

1. Mapa. En la fase de mapa se crean dos tipos de mapeadores, cada uno de los cuales es responsable de procesar una de las tablas. Para cada tupla de la tabla, el

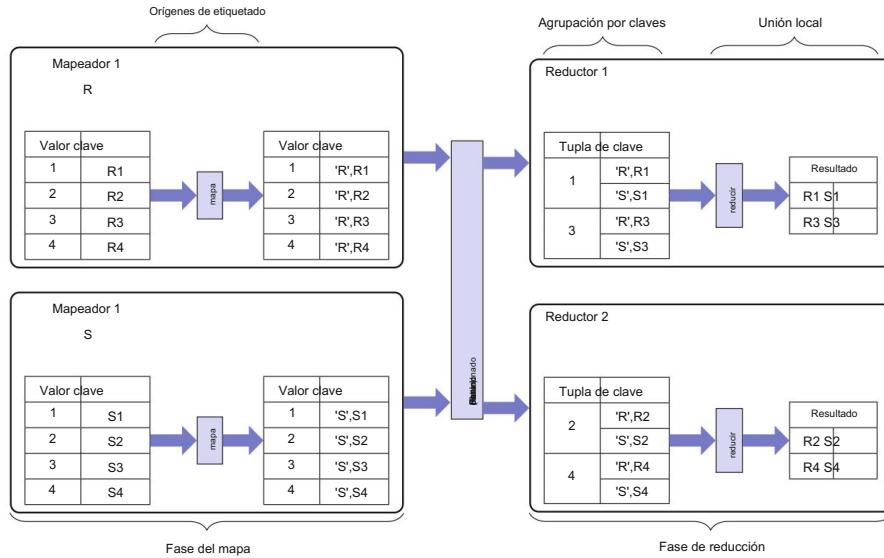


Fig. 10.11 Flujo de datos de unión de repartición

El asignador genera un par clave/valor (k, t, v), donde k es el valor del atributo de unión, v es la tupla completa, y t es la etiqueta que indica la relación de origen de la clave/valor Par. Más específicamente, la fase del mapa consta de los siguientes pasos:

- (a) Escanear los datos de HDFS y generar el par clave/valor.
 - (b) Ordenar la salida del mapa (es decir, el conjunto de pares clave/valor). En el lado del mapa, La salida de cada asignador debe ordenarse antes de barajarse en el reductores.
2. Mezclar. Una vez finalizadas las tareas del mapa, los datos generados se mezclan en el reducir tareas.
3. Reducir. La fase de reducción incluye los siguientes pasos:
- (a) Fusionar. Cada reductor fusiona los datos que recibe mediante la ordenación-fusión. algoritmo. Suponga que la memoria es suficiente para procesar todos los datos ordenados. se ejecutan juntos. Entonces el reductor solo necesita leer y escribir datos en local sistemas de archivos una vez.
 - (b) Unir. Despues de fusionar las ejecuciones ordenadas, el reductor necesita dos fases para Completar la unión. Primero, las tuplas con la misma clave se dividen en dos. partes basadas en la etiqueta que indica su relación de origen. En segundo lugar, las dos partes se unen localmente. Suponiendo que el número de tuplas para la misma clave es pequeño y puede caber en la memoria, este paso solo necesita escanear la ejecución ordenada una vez.
 - (c) Escribir en HDFS. Finalmente, los resultados generados por el reductor deben ser escrito de nuevo en el HDFS.

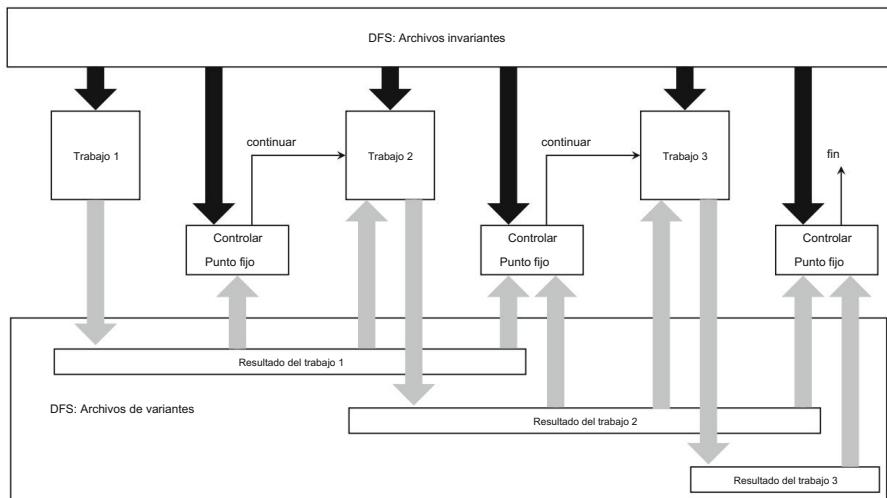


Figura 10.12 Procesamiento de MapReduce para computación iterativa

10.2.2 Procesamiento de datos con Spark

MapReduce básico, como se explicó en la sección anterior, no es adecuado para aplicaciones con uso intensivo de datos que se caracterizan por un cálculo iterativo que requiere una cadena de (es decir, múltiples) trabajos de MapReduce (p. ej., minería de datos) o agregación en línea. En esta sección, analizamos una extensión importante de MapReduce para este tipo de aplicaciones: el sistema Spark. Empezaremos explicando cómo se puede realizar el cálculo iterativo en un sistema MapReduce básico y por qué esto es problemático.

La Figura 10.12 muestra un trabajo iterativo con tres iteraciones que tienen dos características: (1) la fuente de datos de cada iteración consta de una parte variante y una parte invariante: la parte variante consiste en los archivos generados a partir de los trabajos anteriores de MapReduce (flechas grises en la Figura 10.12), y la parte invariante es el archivo de entrada original (flechas negras en la Figura 10.12); (2) podría ser necesaria una comprobación de progreso al final de cada iteración para detectar si se ha alcanzado un punto fijo. El punto fijo tiene diferentes significados en distintas aplicaciones; en el algoritmo de agrupamiento de k-medias que analizamos en el Ejemplo 10.2, puede reflejar si se minimiza la suma de cuadrados dentro del grupo, o en el cálculo de PageRank del Ejemplo 10.4, puede reflejar que el cálculo del rango de cada vértice ha convergido. Esta figura identifica tres aspectos importantes al usar MapReduce para este tipo de tareas. El primero es que, después de cada trabajo (es decir, iteración), los resultados intermedios deben escribirse en el sistema de archivos distribuido (p. ej., HDFS) y learse de nuevo al inicio del siguiente trabajo (iteración). El segundo punto es que no hay garantía de que los trabajos posteriores se asignen a las mismas máquinas. En consecuencia, los datos invariantes que no cambian entre iteraciones no pueden conservarse en los nodos de trabajo y podrían ter

El tercer punto es que se requiere un trabajo adicional al final de cada iteración para comparar los resultados generados entre el trabajo actual y el anterior (es decir, comprobar la convergencia). Todos estos procesos tienen una alta sobrecarga, lo que hace ineficiente el uso de MapReduce para estas aplicaciones. Existen diversos enfoques para abordar estos problemas, algunos específicos de cada tarea, como el análisis de grafos que se analiza en la siguiente sección, mientras que otros, como Spark, son más generales.

Un ejemplo de una carga de trabajo problemática en MapReduce es el algoritmo de agrupamiento k-medias, que se utiliza con frecuencia en el análisis de big data. Presentamos esta carga de trabajo en el Ejemplo 10.2 y analizamos las dificultades para implementarla con MapReduce.

Ejemplo 10.2. El algoritmo k-medias toma un conjunto X de valores y los divide en k grupos. Para ello, coloca cada valor $x_i \in X$ en el grupo C_j cuyo centroide se encuentra a la menor distancia a x_i . El centroide de un grupo es la media de los valores que lo componen. El cálculo de la distancia es la suma de cuadrados dentro del grupo, es decir, $(x_i - \mu_j)^2$, donde μ_j es el centroide del grupo C_j . Por lo tanto, buscamos la asignación que minimice esta función para cada x_i .

El algoritmo k-means estándar toma como entrada el conjunto de valores $X = \{x_1, x_2, \dots, x_r\}$, y un conjunto inicial de centroides $M = \{\mu_1, \mu_2, \dots, \mu_m\}$ (normalmente $r \ll m$) y realiza iterativamente los tres pasos siguientes:

1. Calcular la distancia de cada $x_i \in X$ a cada centroide $\mu_j \in M$ y asignar x_i a clúster C_j si μ_j minimiza la función anterior.
2. Calcule un nuevo conjunto de centroides M de acuerdo con la nueva asignación de valor a racimos.
3. Para cada uno de los clústeres en C , verifique si los valores del centroide nuevo y antiguo son los mismos; si lo son, se ha alcanzado la convergencia y los algoritmos se detienen. De lo contrario, se necesita otra iteración con los nuevos valores del centroide.

La implementación de este algoritmo en MapReduce es sencilla: el primer paso se realiza durante la fase de mapeo, donde cada trabajador (mapeador) realiza el cálculo en un subconjunto de X , mientras que el segundo paso se realiza durante la fase de reducción. El tercer paso, la comprobación de convergencia, es otra tarea, como se mencionó anteriormente. Cabe destacar que todos los mapeadores necesitan el conjunto completo de centroides M ; por lo tanto, la comprobación de convergencia (paso 3) debe difundir los nuevos centroides si no se ha alcanzado la convergencia.

Los problemas con la implementación de trabajos iterativos utilizando MapReduce se exhiben en este ejemplo: los resultados del cálculo al final de cada iteración (es decir, los centroides M recientemente calculados y la configuración actual de los clústeres C) tienen que escribirse en HDFS para que puedan ser leídos por los mapeadores y reductores en la siguiente iteración; dado que la asignación del trabajo de iteración subsiguiente puede ir a cualquier máquina, los datos invariantes (es decir, X) tienen que ser repartidos y leídos nuevamente; y hay un trabajo de verificación de convergencia adicional al final de cada iteración.

Spark soluciona esta deficiencia de MapReduce al proporcionar una abstracción para compartir datos en múltiples etapas de un cálculo iterativo. La abstracción es

Se denomina conjunto de datos distribuido resiliente (RDD). Logra una compartición eficiente de dos maneras: primero, garantiza que las particiones asignadas a cada nodo trabajador se mantengan entre iteraciones para evitar la mezcla de datos; segundo, evita la escritura y lectura desde HDFS entre iteraciones al mantener los RDD en memoria. Dado que la asignación a los trabajadores se mantiene de una iteración a la siguiente, esto es factible.

Un RDD es una estructura de datos que los usuarios pueden crear, decidir cómo se partitiona entre los nodos de trabajo de un clúster y decidir explícitamente si se almacena en disco o en memoria. Si se almacena en memoria, actúa como caché del conjunto de trabajo de la aplicación. Un RDD es una colección inmutable (es decir, de solo lectura) de registros de datos; la actualización de un RDD se realiza mediante una transformación (p. ej., map(), filter(), groupByKey()) que genera un nuevo RDD.

De esta forma, se puede crear un RDD a partir de los datos leídos desde el sistema de archivos o desde otro RDD mediante una transformación.

Ejemplo 10.3 Consideremos la implementación de la agrupación en clústeres de k-medias (Ejemplo 10.2) en Spark. No presentaremos el algoritmo completo aquí, pero sí destacaremos cómo... Aborda las cuestiones:

1. Cree un RDD para los datos invariantes (conjunto X) y guárdelo en la memoria caché para evitar la E/S que se debe realizar entre cada iteración.
2. Cree un RDD para datos variantes (centroideos elegidos M).
3. Calcule la distancia entre $x_i \in X$ y cada $\mu_j \in M$; guarde estas distancias como un RDD D.
4. Cree un nuevo RDD que incluya cada x_i y el μ_j con la distancia mínima desde D.
5. Cree un RDD Mnew que incluya el valor medio de x_i asignado a cada μ_j .
6. Compare M y Mnew y decida si se ha logrado la convergencia (verifique punto fijo).
7. Si aún no ha convergido, entonces $M \leftarrow M_{\text{new}}$. No es necesario recargar los datos invariantes; Spark puede continuar con el paso 10.3.

Un aspecto importante de un RDD es su persistencia a lo largo de las iteraciones (o trabajos de MapReduce). Si se desea esto, se debe aplicar una de las dos transformaciones al RDD: caché o persistencia. Si se desea que el RDD permanezca en la memoria principal a lo largo de los trabajos, se utiliza la caché ; si se requiere flexibilidad para especificar el nivel de almacenamiento (p. ej., solo disco, disco y memoria, etc.), se utiliza la persistencia con las opciones adecuadas, siendo la persistencia en memoria la opción predeterminada.

El cálculo de RDD se realiza de forma perezosa, cuando el programa requiere una acción. Las acciones (por ejemplo, recopilar(), contar()) son diferentes a las transformaciones en que se materializan

²La implementación completa de una variante del algoritmo que analizamos anteriormente se puede encontrar en <https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala> (consultado enero 2018).

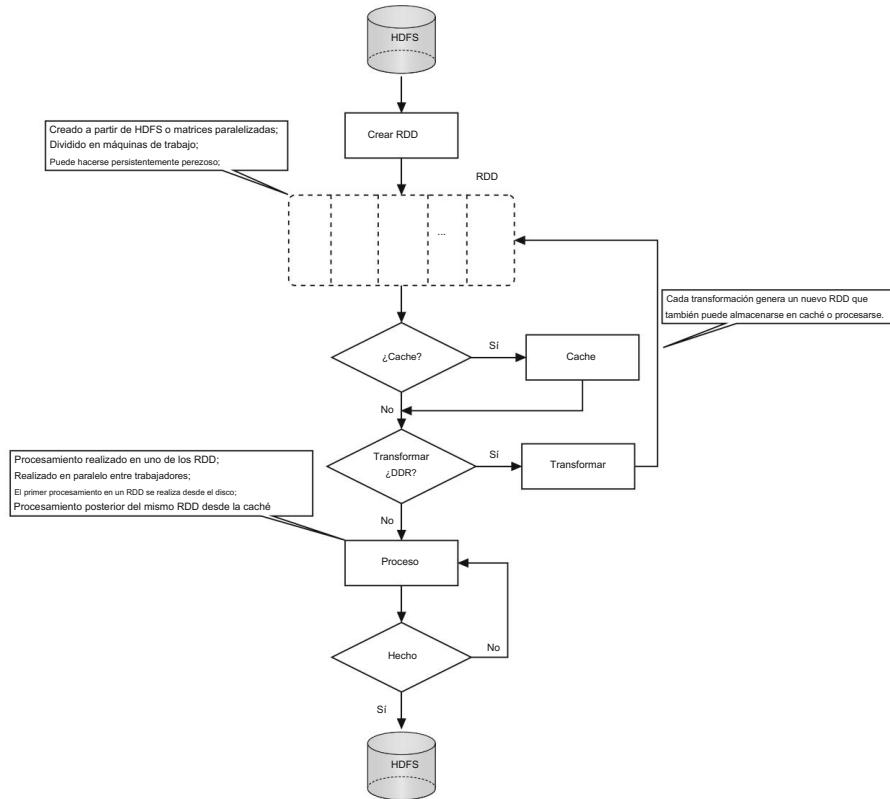


Fig. 10.13 Flujo del programa Spark

El RDD cuando se realiza la primera acción, y la acción especificada se ejecuta en el RDD en todos los nodos donde está particionado. Analizaremos el aspecto de la ejecución más adelante.

Veamos ahora el flujo de trabajo de la ejecución de un programa Spark, que se muestra en la Fig. 10.13. Lo primero que hace el programa es crear un RDD a partir de los datos sin procesar en HDFS. Despues, segun la decisió n del usuario sobre si almacenar en caché o persistir el RDD, el sistema realiza los preparativos necesarios. Posteriormente, puede haber transformaciones adicionales para generar otros RDD y, para cada uno, se especifica la decisió n de almacenar en caché o persistir. Finalmente, el procesamiento comienza con las acciones indicadas en el programa. Como se mencionó anteriormente, la primera acció n en un RDD lo materializa y luego lo aplica. El procesamiento itera sobre múltiples acciones y trabajos.

Analicemos ahora la compatibilidad de Spark con la ejecución de programas escritos con el concepto RDD. Spark espera tener una máquina controladora que ejecute el software del controlador. El controlador genera los RDD indicados en el programa y, tras la primera acción en un RDD, lo materializa, lo divide entre los nodos de trabajo y, a continuación, ejecuta la acción en los trabajadores. El controlador corresponde al maestro.

Nodo en MapReduce mientras el controlador realiza la función de programación. Según la decisión de caché/persistencia para cada RDD, el controlador indica a los trabajadores que tomen la acción apropiada. Cuando los trabajadores indican que la ejecución de la acción ha finalizado, el controlador inicia la acción subsiguiente. Existen optimizaciones habituales con respecto a la gestión de los RDD, la gestión de trabajadores rezagados, etc., pero estas quedan fuera de nuestro alcance.

Spark complementa la tolerancia a fallos estándar de MapReduce al mantener el linaje de los RDD. En otras palabras, mantiene un gráfico de cómo se genera cada RDD a partir de otros. El linaje se construye como un objeto y se almacena de forma persistente para su recuperación. Cuando se produce un fallo y se pierde un RDD, este se puede recalcular según el linaje. Además, como se explica más adelante, cada RDD se partitiona entre máquinas de trabajo, por lo que es probable que la pérdida se limite a algunas particiones de un RDD, y el recálculo se pueda restringir a ellas.

Un objetivo importante de Spark es implementar la arquitectura de referencia que analizamos de forma uniforme, pero proporcionando un ecosistema común. Esto ha resultado en el desarrollo de un SGBD relacional sobre Spark (Spark SQL), un sistema de flujo de datos (Spark Streaming) y un sistema de procesamiento de grafos (GraphX). Analizaremos Spark Stream y GraphX en secciones posteriores.

10.3 Gestión de datos de transmisión

Los sistemas tradicionales de gestión de datos que hemos considerado hasta ahora consisten en un conjunto de objetos desordenados y relativamente estáticos, con inserciones, actualizaciones y eliminaciones menos frecuentes que las consultas. A veces se denominan bases de datos de instantáneas , ya que muestran una instantánea de los valores de los objetos de datos en un momento dado.³ Las consultas sobre estos sistemas se ejecutan en el momento en que se formulan y la respuesta refleja el estado actual de la base de datos. El paradigma típico consiste en ejecutar consultas transitorias sobre datos persistentes .

Ha surgido una clase de aplicaciones que no se ajusta a este modelo de datos ni a este paradigma de consulta. Estas incluyen, entre otras, redes de sensores, análisis de tráfico de red, Internet de las cosas (IoT), indicadores financieros, compras y subastas en línea, y aplicaciones que analizan registros de transacciones (como registros de uso de la web y registros de llamadas telefónicas). En estas aplicaciones, los datos se generan en tiempo real, en forma de una secuencia ilimitada (flujo) de valores. Estas se conocen como aplicaciones de flujo de datos . En esta sección, analizamos los sistemas que las soportan. Las aplicaciones de flujo de datos reflejan la velocidad característica del big data.

Los sistemas que procesan flujos de datos suelen ser de dos tipos: sistemas de gestión de flujo de datos (DSMS), que proporcionan las funcionalidades de un DBMS típico.

3Recuerde que, como explicamos anteriormente, los almacenes de datos suelen almacenar datos históricos para permitir su análisis a lo largo del tiempo. La mayoría de los sistemas que hemos considerado son OLTP, que gestionan instantáneas.

incluido un lenguaje de consulta (declarativo o basado en flujo de datos) y sistemas de procesamiento de flujo de datos (DSPS) que no pretenden incorporar la funcionalidad completa del DBMS.

Los primeros sistemas eran típicamente DSMS, algunos de los cuales utilizaban lenguajes declarativos (p. ej., STREAM, Gigascope, TelegraphCQ), mientras que otros (p. ej., Aurora y su versión distribuida, Borealis) utilizaban un lenguaje de flujo de datos. Los más recientes suelen pertenecer a la clase DSPS (p. ej., Apache Storm, Heron, Spark Streaming, Flink, MillWheel, TimeStream). Muchos de los primeros DSMS eran sistemas monomáquina (excepto Borealis), mientras que los DSPS más recientes son todos sistemas distribuidos/paralelos.

Un supuesto fundamental del modelo de flujo de datos es que los nuevos datos se generan continuamente y en un orden fijo, aunque las tasas de llegada pueden variar según la aplicación, desde millones de elementos por segundo (p. ej., la monitorización del tráfico de Internet) hasta varios elementos por hora (p. ej., las lecturas de temperatura y humedad de una estación meteorológica). El orden de los datos de flujo puede ser implícito (según la hora de llegada al centro de procesamiento) o explícito (según la hora de generación, indicada por una marca de tiempo adjunta a cada elemento de datos por la fuente). Como resultado de estos supuestos, los sistemas de flujo de datos (DSS)⁴ se enfrentan a los siguientes requisitos.

1. Gran parte del cálculo realizado por un DSS se basa en la inserción o en los datos.

Los elementos de flujo recién llegados se introducen en el sistema de forma continua (o periódica) para su procesamiento. Por otro lado, un SGBD tradicional emplea un modelo de cálculo basado principalmente en la extracción o en consultas, donde el procesamiento se inicia al formularse una consulta.

2. Como consecuencia de lo anterior, las consultas y cargas de trabajo DSS suelen ser persistentes (también denominadas consultas continuas, de larga duración o permanentes), ya que se emiten una vez, pero permanecen activas en el sistema posiblemente durante un largo período de tiempo. Esto significa que se debe generar un flujo de resultados actualizados a lo largo del tiempo. Estos sistemas pueden, por supuesto, aceptar y ejecutar consultas ad-hoc transitorias como un SGBD tradicional, pero las consultas persistentes son su característica distintiva.

3. Se supone que un flujo de datos tiene una longitud ilimitada o al menos desconocida.

Por lo tanto, no es posible seguir el enfoque habitual y almacenar los datos por completo antes de ejecutar las consultas; estas deben ejecutarse conforme los datos llegan al sistema. Algunos sistemas emplean un modelo de procesamiento continuo , donde cada nuevo dato se procesa en cuanto llega al sistema (p. ej., Apache Storm, Heron). Otros emplean un modelo de procesamiento en ventanas , donde los datos entrantes se agrupan y procesan por lotes (p. ej., STREAM, Spark Streaming).

Desde la perspectiva del usuario, los datos recién llegados pueden ser más interesantes y útiles, lo que da lugar a la definición de ventanas a nivel de aplicación. Los sistemas que siguen un modelo de procesamiento continuo pueden (y suelen hacerlo) proporcionar ventanas en su API. Por lo tanto, desde la perspectiva del usuario, cumplen ambas funciones. Los sistemas también pueden implementar ventanas internamente para evitar operaciones bloqueantes, como veremos más adelante.

Utilizaremos este término más general cuando la separación entre DSMS y DSPS no sea importante para la discusión.

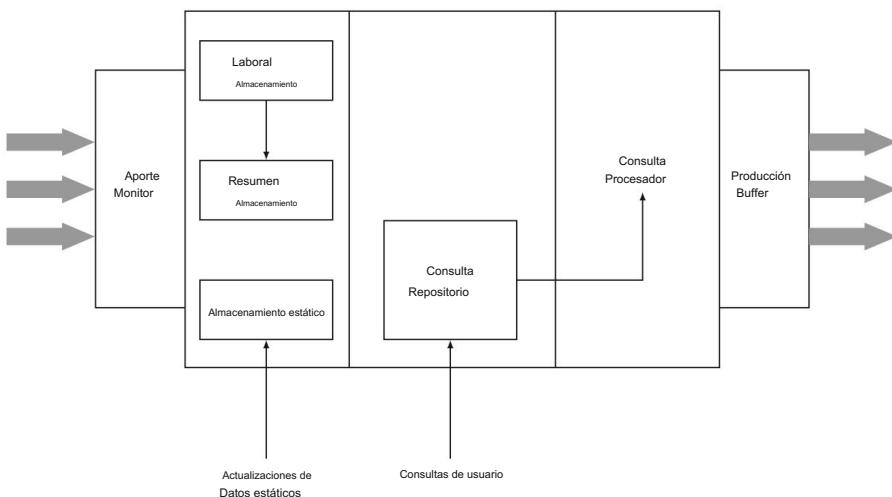


Fig. 10.14 Arquitectura de referencia abstracta para un sistema de gestión de flujo de datos

4. Las condiciones del sistema pueden no ser estables durante la vida útil de una consulta persistente.

Por ejemplo, las tasas de llegada de transmisiones pueden fluctuar y la carga de trabajo de consultas puede cambiar.

En la figura 10.14 se muestra una arquitectura de referencia abstracta de un solo nodo para DSS.

Los datos provienen de una o más fuentes externas. Un monitor de entrada regula la velocidad de entrada, posiblemente descartando elementos si el sistema no puede mantener el ritmo. Los datos se almacenan típicamente en tres particiones: almacenamiento temporal de trabajo (p. ej., para consultas de ventana, que se explicarán en breve), almacenamiento de resumen para sinopsis de flujos (opcional, ya que algunos sistemas no exponen el estado del flujo a las aplicaciones y, por lo tanto, no lo necesitan) y almacenamiento estático para metadatos (p. ej., la ubicación física de cada fuente).

Las consultas de larga duración se registran en el repositorio de consultas y se agrupan para su procesamiento compartido, aunque también se pueden realizar consultas puntuales sobre el estado actual del flujo. El procesador de consultas se comunica con el monitor de entrada y puede reoptimizar los planes de consulta en función de los cambios en las tasas de entrada. Los resultados se transmiten a los usuarios o se almacenan temporalmente en el búfer. Los usuarios pueden refinar sus consultas en función de los resultados más recientes. En un DSS distribuido/paralelo, esta arquitectura se replicaría en cada nodo y se añadirían componentes adicionales para la comunicación y la gestión distribuida de datos.

10.3.1 Modelos de flujo, lenguajes y operadores

Ahora nos centraremos en las cuestiones fundamentales del modelo de los sistemas fluviales. Existe una rica literatura sobre este tema que mencionaremos en las Notas Bibliográficas; nuestro

El objetivo en este punto es resaltar y explicar los conceptos fundamentales para comprender la discusión posterior.

10.3.1.1 Modelos de datos

Un flujo de datos es una secuencia de elementos con marca de tiempo que se anexan y llegan en un orden determinado. Si bien esta es la definición comúnmente aceptada, existen versiones más flexibles; por ejemplo, las tuplas de revisión, que se entiende que reemplazan datos previamente reportados (presumiblemente erróneos), pueden considerarse para que la secuencia no sea de anexión. En los sistemas de publicación/suscripción, donde los datos son producidos por algunas fuentes y consumidos por quienes se suscriben a esas fuentes de datos, un flujo de datos puede considerarse como una secuencia de eventos que se informan continuamente. Dado que los elementos pueden llegar en ráfagas, un flujo puede modelarse como una secuencia de conjuntos (o bolsas) de elementos, donde cada conjunto almacena los elementos que han llegado durante la misma unidad de tiempo (no se especifica ningún orden entre los elementos de datos que han llegado al mismo tiempo). En los modelos de flujo basados en relaciones (p. ej., STREAM), los elementos individuales toman la forma de tuplas relacionales, de modo que todas las tuplas que llegan al mismo flujo tienen el mismo esquema. En los modelos basados en objetos (p. ej., COUGAR y Tribeca), las fuentes y los tipos de elementos pueden ser instancias de tipos de datos jerárquicos con métodos asociados. En sistemas más recientes, como Apache Storm, Spark Streaming y otros, los elementos de datos pueden ser cualquier dato específico de la aplicación; por lo tanto, a veces se utiliza el término genérico "carga útil". Los elementos de flujo pueden contener marcas de tiempo explícitas asignadas por el origen o implícitas asignadas por el DSMS al llegar, por lo que cada elemento de datos es una tupla con marca de tiempo y carga útil. En cualquier caso, el atributo de marca de tiempo puede o no formar parte del esquema del flujo y, por lo tanto, puede o no ser visible para los usuarios. Los elementos de flujo pueden llegar desordenados (si se utilizan marcas de tiempo explícitas) o preprocesados. Por ejemplo, en lugar de propagar el encabezado de cada paquete IP, se puede generar un valor (o varios valores parcialmente preagregados) para resumir la longitud de una conexión entre dos direcciones IP y el número de bytes transmitidos.

Se han definido varias clasificaciones diferentes de modelos de ventanas, pero

Dos criterios son los más importantes y prevalentes:

1. Dirección de movimiento de los puntos finales: Dos puntos finales fijos definen una ventana fija, dos puntos finales deslizantes (hacia adelante o hacia atrás, reemplazando elementos antiguos a medida que llegan elementos nuevos) definen una ventana deslizante, y un punto final fijo y un punto final móvil (hacia adelante o hacia atrás) definen una ventana de referencia.
2. Definición del tamaño de la ventana: Las ventanas lógicas o basadas en tiempo se definen en términos de un intervalo de tiempo, mientras que las ventanas físicas (también conocidas como ventanas basadas en conteo) se definen en términos del número de elementos de datos. Además, las ventanas particionadas pueden definirse dividiendo una ventana en grupos y definiendo una ventana basada en conteo independiente en cada grupo. El tipo más general es una ventana de predicado, en la que un predicado arbitrario especifica el contenido de la ventana; por ejemplo, todos los paquetes de conexiones TCP actualmente abiertas. Una ventana de predicado es

análogas a una vista materializada y también se denominan ventanas de sesión o ventanas definidas por el usuario.

Según esta clasificación, los modelos de ventana más importantes son las ventanas deslizantes basadas en el tiempo y las basadas en el conteo. Estas han atraído la mayor atención y la mayor parte de nuestros análisis se centrarán en ellas.

10.3.1.2 Modelos y lenguajes de consulta de flujo

Un aspecto importante es la semántica de las consultas persistentes (continuas), es decir, cómo generan respuestas. Las consultas persistentes pueden ser monótonas o no monótonas. Una consulta monótona es aquella cuyos resultados se pueden actualizar incrementalmente.

Es decir, basta con reevaluar la consulta con los elementos recién llegados y añadir tuplas que califiquen al resultado. En consecuencia, la respuesta de una consulta persistente monótona es un flujo continuo de resultados que solo se pueden añadir. Opcionalmente, la salida puede actualizarse periódicamente añadiendo un lote de nuevos resultados. Las consultas no monótonas pueden producir resultados que dejan de ser válidos a medida que se añaden nuevos datos y se modifican (o eliminan) los existentes. Por lo tanto, puede que sea necesario volver a calcularlos desde cero en cada reevaluación.

Como se mencionó anteriormente, los DSMS proporcionan un lenguaje de consulta para el acceso. Se pueden identificar dos paradigmas fundamentales de consulta: declarativo y procedimental. Los lenguajes declarativos tienen una sintaxis similar a la de SQL, pero una semántica específica para flujos de datos. Los lenguajes de esta clase incluyen CQL, GSQL y StreaQuel. Los lenguajes procedimentales construyen consultas definiendo un grafo acíclico de operadores (p. ej., Aurora).

Los lenguajes que admiten la ejecución en ventanas proporcionan dos primitivas: tamaño y deslizamiento. La primera especifica la longitud de la ventana y la segunda, su frecuencia de movimiento. Por ejemplo, para una consulta de ventana deslizante basada en tiempo, tamaño=10min, deslizamiento=5sec significaría que nos interesa operar con datos en una ventana de 10 minutos de duración, y que esta se mueve cada 5 segundos. Estos parámetros influyen en la gestión del contenido de la ventana, tema que se aborda en la sección [10.3.2.1](#).

10.3.1.3 Operadores de streaming y su implementación

Las aplicaciones que generan flujos de datos también presentan similitudes en el tipo de operaciones que realizan. A continuación, se enumeran algunas operaciones fundamentales sobre el flujo de datos.

- Selección: Todas las aplicaciones de streaming requieren compatibilidad con filtrado complejo.
- Agregación compleja: Se necesitan agregados complejos, incluidos agregados anidados (p. ej., comparar un mínimo con un promedio móvil), consultas frecuentes de elementos, etc., para calcular tendencias en los datos.

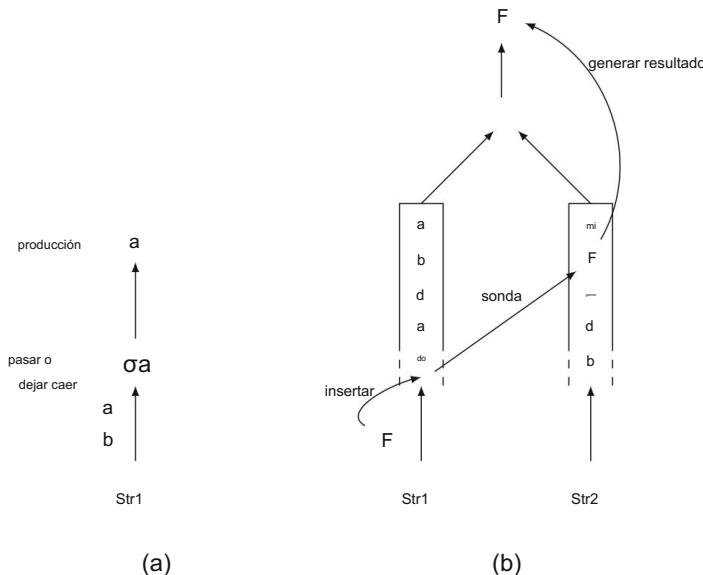


Fig. 10.15 Operadores de consulta continua: (a) Selección, (b) Unión

- Multiplexación y demultiplexación: Es posible que sea necesario descomponer los flujos físicos en una serie de flujos lógicos y, a la inversa, fusionar los flujos lógicos en un solo flujo físico (similar a la agrupación por y la unión, respectivamente).
- Minería de flujos: Se necesitan operaciones como la coincidencia de patrones, la búsqueda de similitudes y la predicción para la minería en línea de datos de flujo.
- Uniones: Se debe incluir compatibilidad con uniones multiflujo y uniones de flujos con metadatos estáticos.

- Consultas con ventana: todos los tipos de consultas anteriores pueden restringirse para devolver resultados dentro de una ventana (por ejemplo, las últimas 24 horas o los últimos cien paquetes).

Si bien estos parecen, en general, operadores de consulta relacional comunes, sus implementación y la optimización presentan nuevos desafíos que analizamos a continuación.

Algunos de estos operadores no tienen estado (p. ej., proyección y selección) y sus implementaciones relacionales pueden utilizarse en consultas de streaming sin modificaciones significativas. La Figura 10.15(a) muestra la implementación del operador de selección como ejemplo. Las tuplas entrantes simplemente se filtran según la condición de selección.

Sin embargo, los operadores con estado (p. ej., las uniones) presentan un comportamiento de bloqueo en sus implementaciones relacionales que no es adecuado para los DSS. Por ejemplo, antes de devolver la siguiente tupla, la unión de bucles anidados (NLJ) podría escanear toda la relación interna y comparar cada tupla con la tupla externa actual. Dada la naturaleza ilimitada de los datos en streaming, este bloqueo es problemático. Se ha demostrado que una consulta es monótona si, y solo si, no es bloqueante, lo que significa que no necesita esperar hasta el marcador de fin de entrada para generar resultados. Algunos operadores

Tienen contrapartes no bloqueantes, como uniones y agregados simples. Por ejemplo, una unión hash simétrica canalizada no bloqueante (de dos flujos de caracteres, Str1 y Str2) genera tablas hash sobre la marcha para Str1 y Str2 (véase la figura 10.15(b)).

Las tablas hash se almacenan en la memoria principal y, cuando llega una tupla de una de las relaciones, se inserta en su tabla y se buscan coincidencias en las demás tablas para generar resultados que involucren la nueva tupla, si las hay. Las uniones de más de dos flujos y las uniones de flujos con una relación estática son extensiones directas. En las primeras, por cada llegada a una entrada, se prueban los estados de todas las demás entradas en cierto orden. En las segundas, las nuevas llegadas al flujo activan el sondeo de la relación.

Dado que mantener tablas hash en flujos sin límites no es práctico, la mayoría de los DSMS solo admiten uniones de ventanas, donde se definen las ventanas sobre cada flujo de entrada y se calculan uniones sobre los datos en estas ventanas en función de la semántica de ventana específica.

Se puede desbloquear un operador de consulta reimplementándolo de forma incremental, limitándolo a operar sobre una ventana y aprovechando las restricciones de flujo, como las puntuaciones, que son restricciones (codificadas como elementos de datos) que especifican las condiciones para todos los elementos futuros. Volveremos a las puntuaciones en breve. Los operadores de ventana deslizante procesan dos tipos de eventos: la llegada de nuevos datos y la expiración de los datos antiguos. Analizaremos esto en detalle en la siguiente sección, donde abordaremos los problemas del procesamiento de consultas.

10.3.2 Procesamiento de consultas sobre flujos de datos

Con algunas modificaciones, la metodología de procesamiento de consultas sobre datos en streaming es similar a su contraparte relacional: las consultas declarativas se traducen en planes de ejecución que asignan los operadores lógicos especificados en la consulta a implementaciones físicas. Sin embargo, surgen varias diferencias en los detalles.

Una diferencia importante es la introducción de consultas persistentes y el hecho de que las operaciones consumen datos introducidos por las fuentes en el plan, en lugar de extraerlos de las fuentes como en un SGBD tradicional. Además, como se mencionó anteriormente, las operaciones pueden ser (y a menudo lo son) más complejas que los operadores relacionales e involucran UDF. Las colas permiten que las fuentes introduzcan datos en el plan de consulta y que las operaciones recuperen datos según sea necesario. Una estrategia de programación simple asigna una franja de tiempo a cada operación, durante la cual se extraen tuplas de sus colas de entrada, se procesan según el orden de la marca de tiempo y se depositan las tuplas de salida en la cola de entrada de la siguiente operación (Fig. 10.16).

Como se mencionó anteriormente, los sistemas de almacenamiento distribuido (DSS) pueden seguir el modelo de procesamiento continuo o el modelo de ejecución en ventanas. Una preocupación fundamental en este último caso es la gestión de las ventanas, concretamente la adición y eliminación de elementos de datos a/desde la ventana actual. Esto representa otra distinción con respecto a los SGBD relacionales, y se aborda en la sección 10.3.2.1. En los sistemas de flujo, que abordamos, surgen otros dos problemas: la gestión de la carga cuando la tasa de llegada de datos supera la capacidad de procesamiento del sistema (sección 10.3.2.2) y el manejo de datos desordenados.

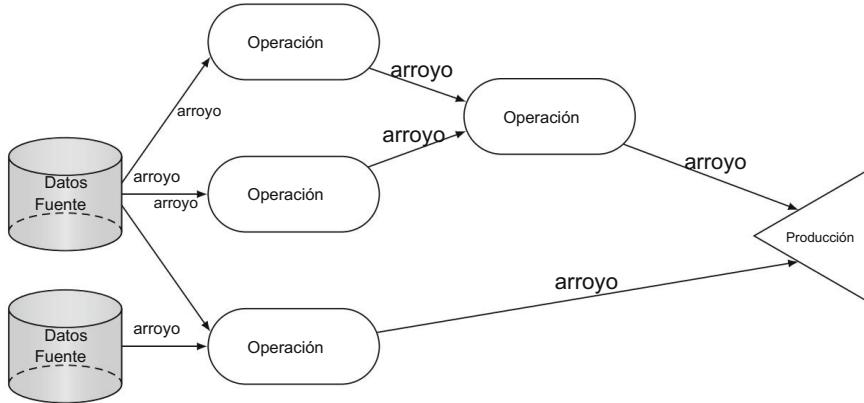


Fig. 10.16 Ejemplo de plan de consulta de flujo

Elementos (Sección 10.3.2.3). Finalmente, las consultas persistentes ofrecen oportunidades adicionales para el procesamiento de múltiples consultas, tema que se aborda en la Sección 10.3.2.4.

Los DSS distribuidos y paralelos siguen caminos diferentes, similares a sus contrapartes relacionales. En el caso distribuido, la técnica fundamental consiste en particionar el plan de consulta entre múltiples nodos de procesamiento de los datos que residen en dichos nodos.

Partitionar el plan de consultas implica asignar operadores de consulta a los nodos y puede requerir un reequilibrio con el tiempo. Los problemas aquí son similares a los de los SGBD distribuidos que hemos analizado en detalle anteriormente en este libro. En sistemas paralelos, se suele seguir el procesamiento de datos en paralelo, donde los datos del flujo se partitionan y cada nodo de procesamiento ejecuta la misma consulta en un subconjunto de los datos. La mayoría de los sistemas modernos siguen este último enfoque, por lo que lo explicamos con más detalle en la Sección 10.3.2.5.

10.3.2.1 Ejecución de consultas en ventana

Anteriormente, se mencionó que, en la ejecución en ventana, el sistema debe gestionar la llegada de nuevos datos y la expiración de los antiguos. Las acciones que se realizan tras la llegada y la expiración varían según el operador. Un nuevo dato puede generar nuevos resultados (p. ej., unión) o eliminar resultados generados previamente (p. ej., negación). Además, un dato expirado puede provocar la eliminación de datos del resultado (p. ej., agregación) o la adición de nuevos datos (p. ej., eliminación de duplicados y negación). Cabe destacar que no se analiza aquí el caso en el que una aplicación elimina un dato. Este análisis se centra en la eliminación de datos de los resultados de una consulta como consecuencia de las operaciones en ventana.

Consideremos, por ejemplo, la unión de una ventana deslizante: un dato recién llegado a una de las entradas prueba el estado de la otra entrada, como en una unión de flujos sin límites.

Además, los datos caducados se eliminan del estado.

La expiración de una ventana individual basada en tiempo es simple: un elemento de datos expira si su marca de tiempo queda fuera del rango de la ventana.

En una ventana basada en conteo, la cantidad de elementos de datos permanece constante a lo largo del tiempo. Por lo tanto, la expiración puede implementarse sobre escribiendo el elemento de datos más antiguo con uno nuevo. Sin embargo, si un operador almacena el estado correspondiente a la salida de una unión de ventana basada en conteo, el número de elementos de datos en el estado puede cambiar según los valores del atributo de unión de las nuevas tuplas.

En general, existen dos técnicas para el procesamiento de consultas de ventana deslizante y el mantenimiento de estado: el enfoque de tupla negativa y el enfoque directo. En el enfoque de tupla negativa, a cada ventana referenciada en la consulta se le asigna un operador que genera explícitamente una tupla negativa por cada expiración, además de insertar las tuplas recién llegadas en el plan de consulta. Por lo tanto, cada ventana debe materializarse para que se produzcan las tuplas negativas adecuadas. Las tuplas negativas se propagan a través del plan de consulta y son procesadas por los operadores de forma similar a las tuplas normales, pero también hacen que los operadores eliminen las tuplas "reales" correspondientes de su estado. El enfoque de tupla negativa puede implementarse eficientemente utilizando tablas hash como estado del operador, de modo que las tuplas expiradas se puedan consultar rápidamente en respuesta a las tuplas negativas. La desventaja es que la consulta debe procesar el doble de tuplas, ya que cada tupla expira eventualmente de su ventana y genera una tupla negativa correspondiente. Además, el plan debe incluir operadores adicionales para generar tuplas negativas a medida que la ventana avanza.

El enfoque directo gestiona consultas sin negación en ventanas temporales. Estas consultas permiten determinar la fecha de expiración de las tuplas base y los resultados intermedios mediante sus marcas de tiempo de expiración, que corresponden a la hora de llegada más la longitud de la ventana. Por lo tanto, los operadores pueden acceder directamente a su estado y encontrar tuplas expiradas sin necesidad de tuplas negativas. El enfoque directo no genera la sobrecarga de las tuplas negativas ni requiere almacenar las ventanas base referenciadas en la consulta. Sin embargo, puede ser más lento que el enfoque de tuplas negativas para consultas en múltiples ventanas, ya que los búferes de estado pueden requerir un escaneo secuencial durante las inserciones o eliminaciones.

10.3.2.2 Gestión de carga

Las tasas de llegada de flujos pueden ser tan altas que no se puedan procesar todas las tuplas, independientemente de las técnicas de optimización (estáticas o en tiempo de ejecución) utilizadas. En este caso, se pueden aplicar dos tipos de deslastre de carga: aleatorio o semántico. Este último utiliza las propiedades del flujo o los parámetros de calidad de servicio para descartar las tuplas que se consideran menos significativas que otras. Como ejemplo de deslastre de carga semántico, considere realizar una unión de ventana deslizante aproximada con el objetivo de alcanzar el tamaño máximo del resultado. La idea es que las tuplas que están a punto de expirar o las que no se espera que produzcan muchos resultados de unión se descarten (en caso de limitaciones de memoria) o se inserten en el estado de unión, pero se ignoren durante el paso de sondeo (en caso de limitaciones de CPU). Tenga en cuenta que son posibles otros objetivos, como obtener una muestra aleatoria del resultado de la unión.

En general, es deseable reducir la carga de forma que se minimice la pérdida de precisión. Este problema se complica cuando se involucran múltiples consultas con muchos operadores, ya que debe decidirse en qué punto del plan de consulta se deben eliminar las tuplas. Claramente, eliminar tuplas al principio del plan es efectivo, ya que todos los operadores posteriores disfrutan de una carga reducida. Sin embargo, esta estrategia puede afectar negativamente la precisión de muchas consultas si se comparten partes del plan. Por otro lado, reducir la carga más adelante en el plan, después de evaluar los subplanes compartidos y de que los únicos operadores restantes sean específicos de consultas individuales, puede tener poco o ningún efecto en la reducción de la carga general del sistema.

Una cuestión que surge en el contexto del deslastre de carga y la generación de planes de consulta es si un plan óptimo elegido sin deslastre de carga sigue siendo óptimo si se utiliza este. Se ha demostrado que esto es así para los agregados de ventana deslizante, pero no para las consultas que incluyen uniones de ventana deslizante.

Tenga en cuenta que, en lugar de descartar tuplas durante períodos de alta carga, también es posible dejarlas de lado (por ejemplo, transferirlas al disco) y procesarlas cuando la carga haya disminuido. Por último, tenga en cuenta que en el caso de reejecución periódica de consultas persistentes, aumentar el intervalo de reejecución puede considerarse como una forma de reducción de carga.

10.3.2.3 Procesamiento fuera de orden

Hasta ahora, en nuestra discusión se ha asumido que el DSS procesa los datos entrantes en orden, generalmente según la marca de tiempo. Sin embargo, esto no siempre es realista. Dado que los datos provienen de fuentes externas, algunos elementos pueden llegar tarde o fuera de orden con respecto a su tiempo de generación. Además, puede que no se reciban datos de una fuente (por ejemplo, un sensor remoto o un enrutador) durante un tiempo, lo que podría significar que no hay nuevos datos que reportar o que la fuente está inactiva. Particularmente en sistemas distribuidos, esto debe considerarse la condición operativa normal debido a desconexiones de red, tiempo de recuperación, etc. Por lo tanto, debe considerarse el procesamiento fuera de orden.

Una primera estrategia para abordar este problema es incorporar un tiempo de holgura que establece un límite superior para el grado de desorden que pueden tener los datos. Aurora utiliza, por ejemplo, un operador de ordenación con búfer donde el flujo entrante se almacena en el búfer durante este tiempo de holgura antes de ser procesado. El operador genera el flujo ordenado según algún atributo. Los datos que llegan después de las unidades de tiempo de holgura se descartan.

Truviso introduce el concepto de "desviación" para adaptarse a los casos en que los flujos de la misma fuente de datos se ordenan entre sí, pero pueden producirse retrasos en la alimentación de datos de algunas fuentes. Cuando el monitor de entrada detecta esto, inicia un período de "desviación" durante el cual almacena en búfer los datos de otras fuentes. La diferencia entre ambos radica en que Aurora puede definir la holgura por operador, mientras que Truviso gestiona la desviación en el monitor de entrada.

Otra solución es usar las puntuaciones introducidas previamente. En este caso, una puntuación es una tupla especial que contiene un predicado que se garantiza que será satisfecho por el resto del flujo de datos. Por ejemplo, una puntuación con la marca de tiempo del predicado > 1262304000 garantiza que no se crearán más tuplas.

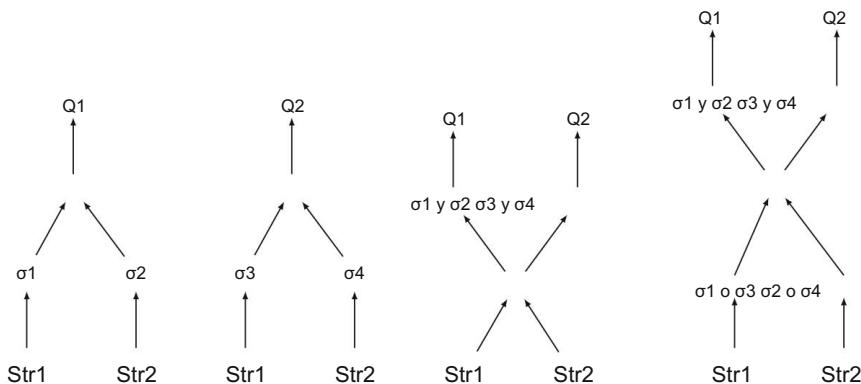


Fig. 10.17 Planes de consulta separados y compartidos para el primer y segundo trimestre

llegan con marcas de tiempo inferiores al tiempo Unix dado; por supuesto, si esta puntuación es generada por la fuente, entonces es útil solo si las tuplas llegan en orden de marca de tiempo. Las puntuaciones que determinan las marcas de tiempo de futuras tuplas normalmente se denominan latidos.

10.3.2.4 Optimización de múltiples consultas

Las consultas de bases de datos pueden compartir partes idénticas, y las técnicas para optimizar un lote de consultas han sido de interés desde hace tiempo, lo que se conoce como optimización multiconsulta. En los sistemas de streaming que admiten consultas persistentes, existe una mayor oportunidad de detectar y explotar componentes y estados compartidos al procesarlos. Por ejemplo, las consultas agregadas con diferentes longitudes de ventana y posiblemente diferentes intervalos de SLIDE pueden compartir estados y estructuras de datos. De igual forma, el estado y el cálculo pueden compartirse entre predicados y uniones similares. Por lo tanto, un DSS puede agrupar consultas similares y ejecutar un único plan de consulta por grupo.

La Figura 10.17 muestra algunos de los problemas que plantean los planes de consulta compartidos. Los dos primeros planes corresponden a la ejecución independiente de las consultas Q1 y Q2, donde las selecciones se evalúan antes de las uniones. El tercer plan ejecuta ambas consultas y evalúa primero la unión y, a continuación, las selecciones (tenga en cuenta que el operador de unión crea dos copias de su flujo de salida). A pesar de compartir el trabajo entre dos consultas, el tercer plan puede ser menos eficiente que la ejecución independiente si solo una pequeña fracción del resultado de la unión satisface los predicados de selección σ_1 a σ_4 . En tal caso, el operador de unión realizará una gran cantidad de trabajo innecesario con el tiempo. El cuarto plan soluciona este problema mediante el prefiltrado de los flujos antes de su unión.

10.3.2.5 Procesamiento de flujo de datos en paralelo

La mayoría de los DSS modernos se ejecutan en clústeres paralelos a gran escala; por lo tanto, se denominan sistemas de procesamiento de flujos de datos paralelos (PDSPS). Estos sistemas presentan similitudes significativas con las bases de datos paralelas que analizamos en el capítulo 8 y, quizás aún más importante, con los marcos de procesamiento de big data de la sección 10.2 de este capítulo. Por lo tanto, en la siguiente discusión nos basamos en dichas discusiones y apelamos a las características específicas de los sistemas de flujo de datos discutidos anteriormente.

El entorno de ejecución típico en estos sistemas se caracteriza por la ejecución paralela de operadores continuos. Como se muestra en la figura 10.16, cada vértice representa una operación continua diferente, asignada a varios nodos de trabajo. Para simplificar el análisis, supongamos que cada trabajador solo ejecuta una operación.

En este contexto, cada máquina de trabajo ejecuta la operación que se le asigna en una partición del flujo de datos y produce resultados que se transmiten a los trabajadores que ejecutan la operación subsiguiente en el plan de consulta. Es importante destacar que la partición del flujo se realiza entre cada par de operaciones.

Así pues, la ejecución de cada operación sigue tres pasos:

1. Particionado del flujo entrante; 2. Ejecución de la operación en la partición; y 3. (Opcionalmente) agregación de los resultados de los trabajadores.

Particionado de flujos

Como en todos los sistemas paralelos, un objetivo particular de la partición es obtener una carga equilibrada entre los trabajadores para evitar rezagos. La característica diferenciadora es que el conjunto de datos asignado a cada trabajador llega en streaming; por lo tanto, la partición de los datos (según un atributo clave) entre múltiples trabajadores debe realizarse sobre la marcha, en lugar de como un proceso sin conexión, como en los sistemas descritos en la sección 10.2.

El enfoque más sencillo para equilibrar la carga en sistemas distribuidos consiste en distribuirla aleatoriamente entre los trabajadores. El particionamiento aleatorio enruta los datos entrantes entre los trabajadores mediante un sistema round-robin (de ahí que también se le conozca como particionamiento round-robin). Este tipo de particionamiento genera una carga de trabajo perfectamente equilibrada. Para aplicaciones sin estado, esto funciona bien, pero las aplicaciones con estado requieren mayor cuidado. Dado que los datos con la misma clave pueden asignarse a diferentes trabajadores, se requiere un paso de agregación para que las operaciones con estado combinen los resultados parciales de cada trabajador para cada clave en cada paso de la ejecución (más información en la sección 10.3.2.5). La agregación es costosa y debe tenerse en cuenta. Además, el particionamiento aleatorio también requiere un alto espacio para las operaciones con estado, ya que cada trabajador debe mantener el estado de cada clave.

El otro extremo es la partición hash, una técnica que hemos visto varias veces. El hash garantiza que los elementos de datos con la misma clave se asignen a un trabajador, eliminando así el costoso paso de agregación y minimizando los requisitos de espacio.

Dado que solo un trabajador mantiene el estado de cada valor de clave, esto puede resultar en una distribución de carga muy desequilibrada, especialmente para flujos de datos sesgados (en términos de valores de clave).

Para aplicaciones con estado, la redistribución y la partición hash constituyen límites superiores para el costo de la división de claves y el desequilibrio de carga, respectivamente. Los trabajos más recientes se han centrado en encontrar algoritmos de partición dentro de estos extremos. Un enfoque prometedor es la división de claves, donde la partición hash se divide posteriormente entre un pequeño número de trabajadores para reducir el desequilibrio de carga. El objetivo es reducir la sobrecarga de la agregación y, al mismo tiempo, el desequilibrio, especialmente en flujos de datos sesgados. El algoritmo de Agrupación Parcial de Claves (PKG) busca reducir el desequilibrio de carga de la partición hash adaptando la división de claves. PKG utiliza la "potencia de dos opciones" al permitir que cada clave se divida entre dos trabajadores. Esto permite a PKG lograr un equilibrio de carga significativamente mejor en comparación con la partición hash y limita el factor de replicación y el costo de agregación. Para datos muy sesgados, PKG se ha ampliado para utilizar más de dos opciones para la cabecera de la distribución.

Aunque se ha demostrado que mejora aún más el equilibrio de carga, su factor de replicación está limitado superiormente por el número de nodos de trabajo en el peor de los casos. Otro enfoque para gestionar datos muy sesgados consiste en utilizar una técnica de partición híbrida donde las tuplas de la cabeza (frecuentes) y la cola (menos frecuentes) de la distribución de claves se tratan de forma diferente, quizás con una preferencia por una asignación equilibrada de las tuplas más importantes desde la cabeza.

Ejecución de carga de trabajo de flujo paralelo

Centrémonos primero en la ejecución de operaciones individuales. Para las operaciones sin estado, no se presentan problemas específicos derivados de la transmisión y el paso de agregación es innecesario. Las operaciones con estado requieren mayor cuidado, y esto es lo que analizamos a continuación.

Si se utiliza el particionamiento aleatorio para una operación con estado, como se mencionó anteriormente, los elementos de datos con la misma clave pueden estar ubicados en diferentes trabajadores, cada uno de los cuales almacenará solo resultados parciales. Por lo tanto, el particionamiento aleatorio requiere un paso de agregación para generar los resultados finales. A modo de ejemplo, la figura 10.18 muestra una operación de conteo en tres trabajadores, donde los diferentes colores indican claves diferentes. Como se puede observar, los elementos de datos con las mismas claves pueden ir a diferentes trabajadores, cada uno de los cuales mantiene el conteo para cada clave (estado) que se agrega al final.

Si se utiliza el particionamiento hash para una operación con estado, todos los elementos de datos con claves idénticas se asignan al mismo trabajador, por lo que no hay un paso de agregación. La Figura 10.19 muestra el particionamiento hash para el mismo ejemplo de conteo que usamos anteriormente.

Al incorporar estos puntos al plan de consulta, cada operación suele tratarse individualmente, y se toman decisiones de partición para cada una, como en Apache Storm y Heron. Por consiguiente, el plan de consulta de ejemplo que se muestra en la Fig. 10.16 adopta la forma de la Fig. 10.20.

Un problema que surge se relaciona con los flujos de datos altamente sesgados. Como se discutió en la sección 10.3.2.5, el enfoque actual comúnmente aceptado es utilizar la división de claves.

Possiblemente con ciertas optimizaciones para datos altamente asimétricos. Otro enfoque es el reparticionamiento de flujos entre operaciones en el plan de consulta. El problema fundamental aquí es redirigir el flujo de datos entre operaciones en el plan de consulta, lo que también requiere la migración de estado (ya que otro trabajador asumirá partes del flujo). Se han desarrollado varias estrategias alternativas, pero el trabajo fundamental en este sentido es Flux, y lo utilizamos como ejemplo para explicar cómo funciona este reparticionamiento. Flux es un operador de flujo de datos que se coloca entre dos operaciones en el plan de consulta; supervisa las cargas de los trabajadores, redirige dinámicamente los datos y migra el estado de un trabajador a otro. Este proceso consta de dos fases: redireccionar los datos y migrar el estado. El redireccionamiento requiere la actualización de las tablas de enrutamiento internas. La migración de estado requiere mayor cuidado, ya que el estado que se mantiene en el trabajador "antiguo" con respecto a esa partición debe serializarse y transferirse al trabajador "nuevo". Esto implica los siguientes pasos: impedir que se acepten nuevas tuplas en la partición; ordenar el estado en el trabajador "antiguo", lo que implica extraer esta información de las estructuras de datos internas; mover el estado al trabajador "nuevo"; desorganizar el estado e instalarlo llenando las estructuras de datos de la máquina "nueva"; y reiniciar la recepción de datos en el flujo.

Esta migración de estado debe ser rápida, obviamente, pero es una tarea compleja que implica protocolos de sincronización complejos. Esta es una de las razones por las que los sistemas modernos no suelen ofrecer compatibilidad integrada para la migración de estado.

10.3.3 Tolerancia a fallos del DSS

La confiabilidad de los DSS distribuidos/paralelos tiene similitud con los DBMS relationales, pero los problemas se ven exacerbados por el hecho de que es necesario lidiar con flujos de datos.

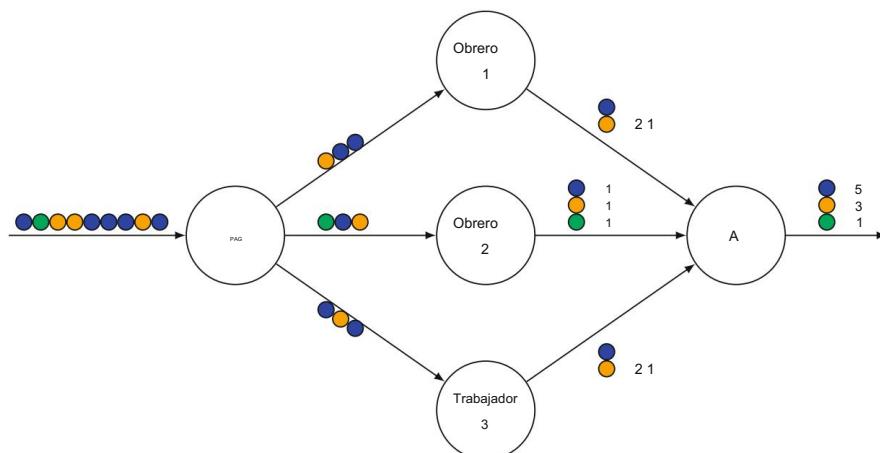


Fig. 10.18 Partición de flujo round-robin

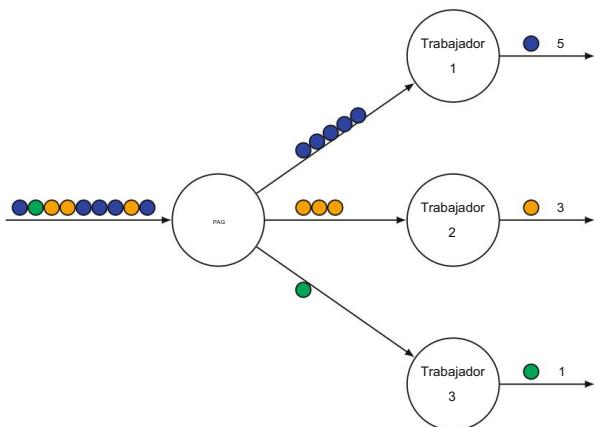


Fig. 10.19 Particionado de flujo basado en hash

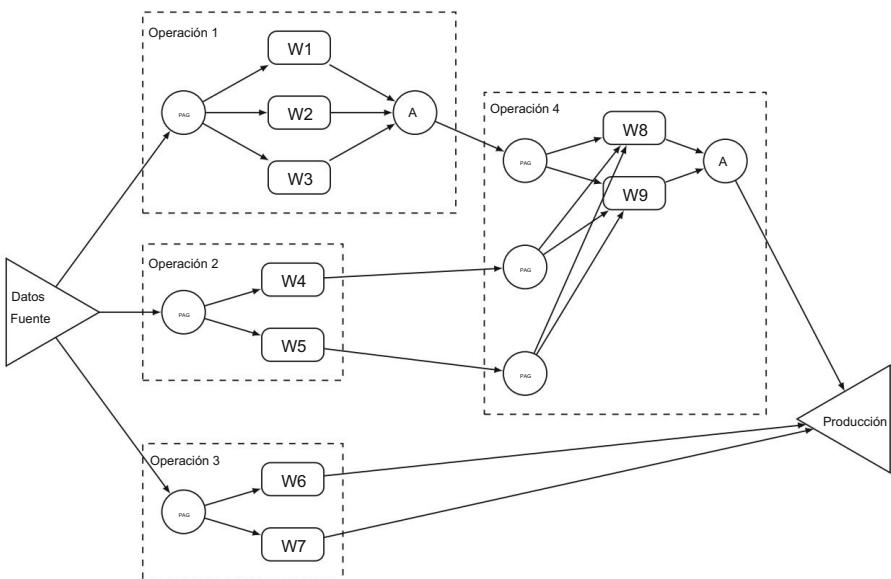


Fig. 10.20 Ejemplo de plan de consulta de flujo paralelo

A través de los planes de consulta. Repasemos primero la distinción que hicimos anteriormente entre los sistemas que partitionan el plan de consulta y ejecutan cada parte en un nodo de servidor diferente y aquellos que partitionan los datos y replican el plan de consulta en cada nodo (es decir, la ejecución paralela de datos). En estos últimos, los fallos se pueden gestionar mediante técnicas de replicación de trabajadores que analizamos en la sección 10.2.1. Sin embargo, en el primer caso, los servidores están "conectados" al ejecutar partes del plan de consulta y los datos fluyen de los servidores ascendentes a los descendentes. Por lo tanto, un fallo en un nodo p

Interrumpir la ejecución de consultas debido a la pérdida de información significativa (transitoria) del estado y la detención de los servidores de bajada que ya no reciben datos. Por consiguiente, estos sistemas necesitan implementar técnicas de alta disponibilidad.

Un tema importante es la semántica de ejecución de consultas que proporciona un sistema a medida que los datos fluyen a través de la red de servidores (o a través del plan de consultas). Hay tres alternativas: al menos una vez, como máximo una vez y exactamente una vez. La semántica de al menos una vez (también llamada recuperación de reversión) indica que el sistema garantiza procesar cada elemento de datos al menos una vez, pero no garantiza los duplicados. Por lo tanto, si un nodo fallido reenvía nuevamente un elemento de datos después de la recuperación, ese elemento de datos puede procesarse nuevamente y producir una salida duplicada. Por el contrario, si se adopta la semántica de como máximo una vez (también llamada recuperación de brecha), el sistema garantiza que los elementos de datos duplicados se detectarán y no se procesarán, pero es posible que ciertos elementos de datos no se ejecuten en absoluto. Esto puede deberse a la gestión de carga, como se mencionó anteriormente, o como resultado de la falla de un nodo que ignora, al recuperarse, todos los elementos de datos que podría haber recibido mientras estaba inactivo. Finalmente, la semántica de "exactamente una vez" (también llamada recuperación precisa) implica que el sistema ejecuta cada elemento de datos exactamente una vez, por lo que nada se descarta ni se ejecuta más de una vez. Obviamente, cada una de estas funciones requiere una funcionalidad diferente del sistema. Todas son compatibles con los sistemas existentes: Apache Storm y Heron ofrecen a las aplicaciones la opción de "al menos una vez" y "como máximo una vez", mientras que Spark Streaming, Apache Flink y MillWheel aplican la semántica de "exactamente una vez".

Los tipos de técnicas de recuperación para DSS se pueden clasificar en dos enfoques principales: replicación y respaldo ascendente. En el caso de la replicación, por cada nodo que ejecuta una parte del plan de consulta, hay un nodo de réplica que también es responsable de esa parte. Se trata de una configuración principal-secundaria, donde el nodo principal gestiona el plan de consulta mientras esté operativo, y el nodo secundario retoma el trabajo si el principal falla. La configuración entre ambos podría ser de reserva activa, donde tanto el principal como el secundario obtienen datos de los nodos ascendentes y los procesan simultáneamente, enviando solo el principal los resultados descendentes, o de reserva pasiva, donde el principal envía periódicamente la diferencia de estado al secundario y este actualiza su estado en consecuencia. Ambos enfoques pueden complementarse con puntos de control para acelerar la recuperación. Este enfoque se ha propuesto como parte del operador Flux mencionado anteriormente y se utiliza en Borealis. La otra alternativa es el respaldo ascendente, donde los nodos ascendentes almacenan en búfer los datos que fluyen a los nodos descendentes hasta que se procesan. Si un nodo descendente falla y se recupera, obtiene los datos almacenados en el búfer de su nodo ascendente y los reprocesa. Una dificultad de diseño radica en determinar el tamaño de estos búferes para alojar los datos recopilados durante el fallo y la recuperación. Esto es complejo porque se ve afectado por la tasa de llegada de datos, así como por otras consideraciones. Sistemas como Apache Storm y TimeStream adoptan este enfoque.

10.4 Plataformas de análisis de gráficos

Los datos de grafos adquieren una importancia creciente en numerosas aplicaciones. En esta sección, analizamos los grafos de propiedades, que son grafos con atributos asociados a vértices y aristas. Otro tipo de grafo es el grafo RDF (grafo de Marco de Descripción de Recursos), que se describe en el capítulo 12.⁵ Los grafos de propiedades se utilizan para modelar entidades y relaciones en diversos ámbitos, como la bioinformática, la ingeniería de software, el comercio electrónico, las finanzas, el comercio y las redes sociales. Un grafo $G = (V, E, Dv, DE)$ se define mediante un conjunto de vértices V y un conjunto de aristas E , Dv y DE , que se definen a continuación.

⁵

Las características distintivas de los gráficos de propiedades son las siguientes:

Cada vértice del grafo representa una entidad, y cada arista entre un par de vértices representa una relación entre esas dos entidades. Por ejemplo, en un grafo de red social que representa Facebook, cada vértice podría representar un usuario y cada arista podría representar la relación de "amistad". Es posible tener múltiples aristas entre un par de vértices, cada una representando una relación diferente; estos grafos se denominan comúnmente multigrafos. Las aristas pueden tener pesos asignados (grafos ponderados), donde el peso...

de un borde podría tener diferente semántica en diferentes gráficos.

Los grafos pueden ser dirigidos o no dirigidos. Por ejemplo, el grafo de Facebook normalmente no es dirigido y representa la relación de amistad simétrica entre dos usuarios: si el usuario A es amigo del usuario B, entonces el usuario B es amigo del usuario A. Sin embargo, un grafo de Twitter, donde las aristas representan la relación de "seguidores", es dirigido, lo que representa que el usuario A sigue al usuario B, pero la inversa puede no ser necesariamente cierta. Como recordará, los grafos RDF son dirigidos por definición. Como se mencionó, cada vértice y cada arista pueden tener un conjunto de atributos (propiedades) para codificar las propiedades de la entidad (en el caso del vértice) o la relación (en el caso de la arista). Si las aristas tienen propiedades, estos grafos generalmente se denominan grafos etiquetados por arista. DV y DE en la definición de grafo dada anteriormente representan el conjunto de propiedades de vértice y arista, respectivamente. Cada vértice/arista puede tener diferentes propiedades, y cuando nos referimos a las propiedades del grafo en general, escribiremos D o DG .

Los gráficos de la vida real que son objeto de análisis de gráficos (como los gráficos de redes sociales, los gráficos de redes de carreteras, así como los gráficos web que analizamos anteriormente) tienen una serie de propiedades que son importantes y afectan muchos aspectos del diseño del sistema:

1. Estos grafos son muy grandes, algunos con miles de millones de vértices y aristas. Procesar grafos con esta cantidad de vértices y, especialmente, de aristas, requiere cuidado.
2. Muchos de estos gráficos se conocen como gráficos de ley de potencia o gráficos libres de escala en los que hay una variación significativa en los grados de los vértices (conocidos como distribución de grados).

⁵En esta sección, cuando sea necesario, escribiremos VG y EG para referirnos específicamente a los vértices y aristas, respectivamente, del grafo G , pero, omitiremos los subíndices cuando sea obvio.

Sesgo). Por ejemplo, mientras que el grado de vértice promedio en un grafo de Twitter es de 35, los "supernodos" en ese grafo tienen un grado máximo de 2,9 millones.^{6 3}

Siguiendo el punto anterior, el grado de vértice promedio en muchos grafos del mundo real es bastante alto con núcleos de alta densidad. Por ejemplo, el grado de vértice promedio en el grafo de Friendster es de aproximadamente 55 y en el grafo de Facebook es de 190.

4. Algunos grafos del mundo real tienen diámetros muy grandes (es decir, el número de saltos entre dos vértices más lejanos). Estos incluyen los grafos espaciales (p. ej., los grafos de redes de carreteras) y los grafos de red: los diámetros de los grafos de red pueden ser de cientos, mientras que algunas redes de carreteras son mucho mayores. El diámetro del grafo afecta a los algoritmos de análisis de grafos que dependen de visitar y calcular iterativamente cada vértice (lo explicaremos más adelante).

La ejecución eficiente de cargas de trabajo en estos grafos es esencial para las plataformas de big data. Al igual que muchos frameworks de big data, estas son principalmente plataformas paralelas/distribuidas (de escalabilidad horizontal) que dependen de la partición del grafo de datos entre nodos de un clúster o sitios de un sistema distribuido.

Las cargas de trabajo de grafos se suelen dividir en dos clases. La primera son las consultas analíticas (o cargas de trabajo analíticas), cuya evaluación suele requerir el procesamiento de cada vértice del grafo a lo largo de múltiples iteraciones hasta alcanzar un punto fijo. Ejemplos de cargas de trabajo analíticas incluyen el cálculo de PageRank (véase el Ejemplo 10.4), la agrupación en clústeres, la búsqueda de componentes conectados (véase el Ejemplo 10.5) y numerosos algoritmos de aprendizaje automático que utilizan datos de grafos (p. ej., la propagación de creencias). Nos centramos en los diferentes enfoques computacionales desarrollados para estas tareas y en los sistemas que los soportan. Estas son las plataformas de computación iterativa especializadas a las que nos referimos al analizar Spark en la sección anterior. En esta sección, nos centraremos en ellas. La segunda clase de cargas de trabajo son las consultas en línea (o cargas de trabajo en línea), que no son iterativas y suelen requerir acceso a una parte del grafo, y cuya ejecución puede ser asistida por estructuras de datos auxiliares correctamente diseñadas, como los índices. Ejemplos de cargas de trabajo en línea son las consultas de accesibilidad (p. ej., si un vértice de destino es accesible desde un vértice de origen dado), la ruta más corta de un solo origen (encontrar la ruta más corta entre dos vértices) y la correspondencia de subgrafos (isomorfismo de grafos). Posponemos el tratamiento de estas cargas de trabajo al capítulo 11, donde analizamos los SGBD de grafos.

Ejemplo 10.4. PageRank es un algoritmo bien conocido para calcular la importancia de las páginas web. Se basa en el principio de que la importancia de una página está determinada por la cantidad y calidad de otras páginas que apuntan a ella. La calidad, en este caso, se mide como el PageRank de una página (de ahí la definición recursiva). Cada página web se representa como un vértice en el grafo web (véase la figura 10.21), y cada arista dirigida representa una relación de "apuntar a". Por lo tanto, el PageRank de una página web P_i , denotado como $PR(P_i)$, es la suma del PageRank de todas las páginas P_j que apuntan a ella, normalizado por la cantidad de páginas a las que apunta cada P_j . La idea es que si

⁶Advertimos que estos valores cambian a medida que los gráficos evolucionan con el tiempo. Deben considerarse indicativos del punto que planteamos, no como valores definitivos.

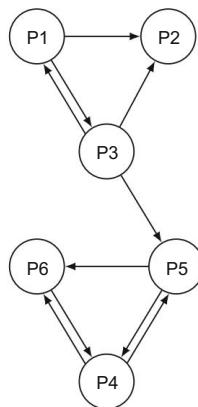


Figura 10.21 Representación gráfica web para el cálculo de PageRank

una página P_i apunta a n páginas (una de las cuales es P_i), su PageRank contribuye a la Cálculo del PageRank de n páginas por igual. La fórmula del PageRank también incluye... factor de amortiguamiento basado en la teoría de paseos aleatorios: si un usuario comienza desde una página web página y continúa haciendo clic en los enlaces para llegar a otras páginas web, este "paseo" le permitirá Finalmente, se detendrá. Por lo tanto, cuando el usuario esté en la página P_i , existe una probabilidad d de que El usuario continuará haciendo clic y $(1 - d)$ la caminata se detendrá; el valor típico Para d es 0,85, determinado como resultado de estudios empíricos. Por lo tanto, si el conjunto de vecinos internos de P_i es B_{P_i} (estos son los enlaces hacia atrás de P_i), y el conjunto de vecinos externos es F_{P_i} (enlaces hacia adelante), la fórmula de PageRank es

$$PR(P_i) = (1 - d) + d \frac{\sum_{j \in F_{P_i}} PR(P_j)}{|F_{P_i}|}.$$

Analizamos PageRank con más detalle en el capítulo 12 cuando consideramos los datos web. gestión. Por el momento, nos centraremos simplemente en el cálculo de la valores usando la Fig. 10.21 como ejemplo. Consideremos la página P_2 ; el PageRank de Esta página es $PR(P_2) = 0,15 + 0,85()$. Claramente, se trata de una función recursiva. fórmula ya que depende del cálculo de los valores de PageRank para P_1 y P_3 . El cálculo generalmente comienza asignando a cada vértice valores de PageRank iguales. (en este caso 1/6 ya que hay 6 vértices), e itera para calcular los valores de cada nodo hasta que se alcanza un punto fijo (es decir, los valores ya no cambian). Por lo tanto, El cálculo de PageRank exhibe ambas propiedades que identificamos para el análisis Cargas de trabajo: cálculo iterativo y participación de cada vértice en cada iteración.⁷

⁷Se han desarrollado varias optimizaciones para el cálculo de PageRank, pero Ignorarlos en esta discusión.

Ejemplo 10.5 Como segundo ejemplo, consideremos el cálculo de los componentes conexos de un grafo. Primero, algunos conceptos básicos. Se dice que un grafo es conexo si hay un camino entre cada par de vértices. Un subgrafo conexo maximal del grafo se llama componente conexo: cada vértice en este componente es accesible desde todos los demás vértices. Encontrar el conjunto de componentes conexos en un grafo es un problema importante de análisis de grafos que se puede usar para varias aplicaciones, como la agrupación en clústeres. Si el grafo es dirigido, entonces un subgrafo donde hay un camino dirigido desde cada par de vértices en ambas direcciones (es decir, un camino del vértice v al u y un camino de u al v) se llama componente fuertemente conexo. Por ejemplo, en la figura 10.21, $\{P_1, P_3\}$ y $\{P_4, P_5, P_6\}$ son dos componentes fuertemente conexos. Si todas las aristas dirigidas en este grafo se reemplazan por aristas no dirigidas, y luego se determinan los componentes conexos maximalistas, esto produce el conjunto de componentes débilmente conexos. Todo el gráfico de la figura 10.21 es un componente débilmente conectado (en este caso, se dice que el gráfico está débilmente conectado).

Encontrar un componente débilmente conectado es un algoritmo iterativo que utiliza DFS de Primera búsqueda (DFS). Dado un grafo $G = (V, E)$ para cada $v \in V$, profundidad uno para determine el componente en el que se encuentra v .

10.4.1 Particionado de gráficos

Como se mencionó anteriormente, la mayoría de los sistemas de análisis de grafos son paralelos, lo que requiere que el grafo de datos se divida y se asigne a nodos de trabajo. Ya abordamos la partición de datos, considerándola en el contexto de sistemas relacionales distribuidos en el capítulo 2 y en el contexto de sistemas de bases de datos paralelas en el capítulo 8. La partición de grafos es diferente debido a las conexiones entre los vértices; esto es, en cierto sentido, similar a preocuparse por las restricciones de integridad entre fragmentos en el diseño de distribuciones, pero los grafos requieren mayor cuidado debido a la intensa comunicación entre vértices, como se explicará en las siguientes secciones. Por lo tanto, se han diseñado algoritmos especiales para la partición de grafos, y la literatura sobre este tema es muy rica.

La partición de grafos puede seguir el enfoque de corte de arista (también conocido como vértice disjunto) o el enfoque de corte de vértice (también llamado arista disjunto). En el primero, cada vértice se asigna a una partición, pero las aristas pueden replicarse entre particiones si conectan vértices de contorno. En el segundo, cada arista se asigna a una partición, pero los vértices pueden replicarse entre particiones si son incidentes a aristas asignadas a particiones diferentes. En ambos enfoques, se persiguen tres objetivos: (1) asignar cada vértice o arista a particiones de forma que estas sean mutuamente excluyentes, (2) asegurar que las particiones estén equilibradas, y (3) minimizar los cortes (ya sean de arista o de vértice) para minimizar la comunicación entre las máquinas a las que se asigna cada partición. Equilibrar estos requisitos es la parte difícil; si, por ejemplo, solo tuviéramos que preocuparnos por equilibrar la carga de trabajo mientras obteníamos particiones mutuamente excluyentes, una asignación round-robin de vértices (o aristas) a las máquinas podría ser suficiente. Sin embargo, no hay garantía de que esto no cause un gran número de cortes.

La partición puede formularse como un problema de optimización de la siguiente manera. Dado un grafo $G(V, E)$ (ignorando las propiedades por el momento), se busca obtener una partición $P = \{P_1, \dots, P_k\}$ de G en k particiones donde los tamaños de P_i estén equilibrados, lo cual puede formularse como el siguiente problema de optimización:

$$\text{minimizar } C(P)$$

sujeto a:

$$w(P_i) \leq \beta \quad \frac{\sum_{j=1}^k w(P_j)}{k}, \quad i \in \{1 \dots k\}.$$

donde $C(P)$ representa el coste total de comunicación de la partición, y $w(P_i)$ es la sobrecarga abstracta de procesar la partición P_i . Los dos enfoques (corte de vértice y corte de arista) difieren en la definición de $C(P)$ y $w(P_i)$, como se explica a continuación. En la formulación anterior, β se introduce como un parámetro de holgura para permitir la partición que no está exactamente equilibrada; si $\beta = 1$, entonces la solución es una partición exactamente equilibrada, y el problema se conoce como problema de optimización de partición de grafos k -equilibrados; si $\beta > 1$, entonces se permite cierta desviación del equilibrio exacto, y esto se conoce como problema de optimización de partición de grafos (k, β) -equilibrados.

Se ha demostrado que este problema es NP-difícil y los investigadores han propuesto métodos heurísticos para lograr una solución aproximada.

La heurística del enfoque de corte de aristas (vértice disjunto) busca lograr una asignación equilibrada de vértices a las particiones, minimizando al mismo tiempo los cortes de aristas; por lo tanto, cada P_i contiene un conjunto de vértices. En estos enfoques, $w(P_i)$ se define en términos del número de vértices por partición (es decir, $w(P_i) = |P_i|$), mientras que el coste de comunicación se calcula como una fracción de los cortes de aristas:

$$C(P) = \frac{\sum_{i=1}^k |e(P_i, V \setminus P_i)|}{|M|}.$$

donde $|e(P_i, P_j)|$ es el número de aristas entre las particiones P_i y P_j .

El algoritmo heurístico de vértices disjuntos más conocido es METIS, que proporciona Partición casi óptima. Consta de tres pasos:

1. Dado un grafo $G_0 = (V, E)$, se produce una jerarquía de grafos sucesivamente engrosados G_1, \dots, G_n tales que $|V(G_i)| > |V(G_j)|$ para $i < j$. Existen varias maneras de engrosar, pero la más popular es la llamada contracción, donde un conjunto de vértices en G_i se reemplaza por un único vértice en G_j ($i < j$). El engrosamiento suele detenerse cuando G_n es lo suficientemente pequeño como para que aún se pueda aplicar un algoritmo de partición de alto costo. Un grafo G_i se engrosa a G_{i+1} al encontrar la coincidencia máxima, que es el conjunto de aristas donde ninguna de sus dos aristas comparte un vértice. Los extremos de cada una de estas aristas se representan mediante un vértice en G_{i+1} .

2. Gn se partitiona utilizando algún algoritmo de partición; como se señaló anteriormente, Gn ahora debería ser lo suficientemente pequeño como para usar cualquier algoritmo de partición deseado, independientemente de su costo computacional.
3. Gn se reduce iterativamente a G0, y en cada paso:
 - (a) la solución de partición en el grafo Gj se proyecta al grafo Gj-1 (nótese que el subíndice más pequeño indica una granularidad más fina del grafo) y
 - (b) la partición de Gj-1 se mejora mediante varias técnicas.

Si bien METIS y los algoritmos relacionados mejoran considerablemente los tiempos de procesamiento de la carga de trabajo de análisis de grafos, no son prácticos ni siquiera para grafos de tamaño mediano debido a su elevado coste computacional. La sobrecarga de particionamiento es un factor importante en los sistemas de análisis de grafos, ya que el tiempo de carga y particionamiento de un grafo puede representar una gran parte del tiempo de procesamiento.

Una simple heurística de partición de vértices disjuntos, basada en hash, se incorpora al repertorio de la mayoría de los sistemas de análisis de grafos que analizaremos a continuación. En este caso, se asigna un vértice a la partición a la que se aplica el hash de su identificador. Esto es simple y muy rápido, y funcionaría razonablemente bien para equilibrar la carga en grafos con una distribución de grados uniforme. Sin embargo, en grafos reales con sesgo de grados, como se mencionó anteriormente, el resultado puede ser una carga de trabajo desequilibrada. Los modelos de partición por corte de aristas distribuyen la carga en términos de vértices, pero para algunos algoritmos la carga es proporcional al número de aristas, lo cual no estaría equilibrado en grafos sesgados. Para estos casos, serían adecuadas heurísticas más sofisticadas que presten atención a la estructura del grafo.

Un enfoque de este tipo es la propagación de etiquetas, donde cada vértice comienza con su propia etiqueta, que intercambia iterativamente con sus vecinos. En cada iteración, cada vértice asume la etiqueta más frecuente de su entorno; cuando las frecuencias son idénticas, se utiliza un método para seleccionar la etiqueta. Este proceso iterativo se detiene cuando las etiquetas de los vértices dejan de cambiar. Esta técnica es sensible a la estructura del grafo, pero no garantiza una partición equilibrada. Una forma de lograr el equilibrio es comenzar con una partición no equilibrada y luego utilizar un algoritmo voraz de propagación de etiquetas para reubicar los vértices y lograr el equilibrio (o casi el equilibrio). El algoritmo voraz mueve los vértices para maximizar una función de utilidad de reubicación sujeta a restricciones de equilibrio. La función de utilidad podría ser, por ejemplo, el número de vecinos del grafo que estarán en la misma partición. Es posible combinar METIS con la propagación de etiquetas incorporando esta última en la fase de engrosamiento. Nuevamente, el problema se modela como un problema de partición restringida que maximiza una función de utilidad que presta atención a la vecindad del vértice para minimizar los cortes de aristas.

Ejemplo 10.6 Considere el grafo de la Fig. 10.22a. En la Fig. 10.22(b) se muestra una partición disjunta de este grafo, donde los cortes de aristas se representan con líneas discontinuas. Esta partición se logró mediante el algoritmo hash descrito anteriormente. Observe que esto provoca que se corten 10 de las 12 aristas totales. Este ejemplo demuestra la dificultad de particionar grafos con vértices de alto grado (en este grafo, los vértices v3 y, en particular, v4 son de alto grado), lo que resulta en cortes de aristas altos. METIS funciona mejor en este grafo, pero no puede generar tres particiones y, en su lugar, produce

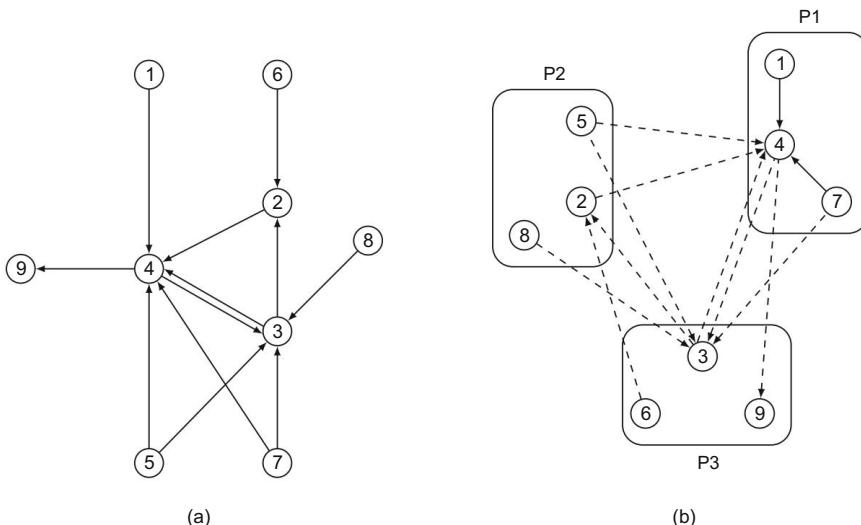


Fig. 10.22 Ejemplo de partición. (a) Ejemplo de gráfico. (b) Partición disjunta de vértices (corte de aristas)

dos: $\{v_1, v_3, v_4, v_7, v_9\}$ y $\{v_2, v_5, v_6, v_8\}$, lo que da como resultado cinco cortes de arista (una partición bidireccional basada en hash da como resultado 8 cortes de arista).

Se ha demostrado que las heurísticas de corte de aristas funcionan bien para grafos con vértices de bajo grado, pero funcionan mal en grafos de ley de potencia que causan un alto número de cortes de aristas. METIS se ha modificado para abordar este problema en particular, pero su rendimiento al tratar con grafos muy grandes sigue siendo un problema. Se acepta generalmente que los enfoques de corte de vértices que asignan aristas a particiones individuales mientras partitionan (replican) los vértices incidentes a estas aristas manejan grafos de ley de potencia con mayor facilidad (es decir, cada P_i contiene un conjunto de aristas). La definición de $w(P_i)$, en este caso, es el número de aristas en la partición P_i , es decir, $w(P_i) = |e(P_i)|$. Para estas heurísticas, la comunicación $C(P)$ se verá afectada por el factor de replicación de cada vértice (definido como el número de particiones a las que se asigna ese vértice); esto se puede formular de la siguiente manera:

$$C(P) = \frac{|A(v)| v - V}{|V|}.$$

donde $A(v) \subseteq \{P_1, \dots, P_k\}$ representa el conjunto de particiones en las que se asigna el vértice v .

El hash también es una opción como heurística de corte de vértices: en este caso, el hash se aplica a los identificadores de los dos vértices que inciden en una arista. Es simple, rápido (ya que se puede parallelizar fácilmente) y proporcionaría una buena partición equilibrada. Sin embargo, puede provocar una alta replicación de vértices. Es posible usar el hash, pero se debe controlar la

factor de replicación. Un enfoque que se ha propuesto es definir, para la arista eu,v , conjuntos de restricciones C_u y C_v , respectivamente, para sus vértices incidentes u y v . Estos son los conjuntos de particiones sobre los que u y v pueden replicarse. Obviamente, la arista eu,v tiene que asignarse a una partición que sea común a los conjuntos de restricciones de u y v , es decir, $C_u \cap C_v$. La restricción establece un límite en el número de particiones a las que se puede asignar un vértice, controlando así el límite superior del factor de replicación. Una forma de generar estos conjuntos de restricciones es definir una matriz cuadrada de particiones y asignar como S_u (de manera similar a S_v) mediante el hash de u (de manera similar a v) a una de las particiones (por ejemplo, P_i), y tomando las particiones que se encuentran en la misma fila y columna que P_i .

También se han diseñado heurísticas de corte de vértice que tienen en cuenta las características del grafo. Un algoritmo voraz decide cómo asignar la $(l+1)$ -a arista a una partición de forma que se minimice el factor de replicación. Por supuesto, la asignación de la $(l+1)$ -a arista depende de la asignación de las primeras R aristas, por lo que el historial es importante.

La ubicación del borde eu,v se decide utilizando las siguientes reglas heurísticas:

1. Si la intersección de $A(u)$ y $A(v)$ no está vacía (es decir, hay algunas particiones que contienen tanto u como v), entonces asigne eu,v a una de las particiones en la intersección.
2. Si la intersección de $A(u)$ y $A(v)$ está vacía, pero si $A(u)$ y $A(v)$ no están individualmente vacíos, entonces asigne eu,v a una de las particiones en $A(u) \cup A(v)$ con la mayor cantidad de aristas sin asignar.
3. Si solo uno de $A(u)$ y $A(v)$ no está vacío (es decir, solo uno de u o v ha sido asignado a particiones), entonces asigne eu,v a una de las particiones del vértice asignado.
4. Si tanto $A(u)$ como $A(v)$ están vacías, entonces asigne eu,v a la partición más pequeña.

Este algoritmo considera la estructura del grafo, pero es difícil de paralelizar para obtener un alto rendimiento, ya que se basa en el historial. La paralelización requiere mantener un estado global actualizado periódicamente o una aproximación donde cada máquina solo considera su historial de estado local sin mantener uno global.

También es posible utilizar métodos de corte de vértices y de corte de aristas dentro de un mismo algoritmo de partición. PowerLyra, por ejemplo, utiliza un algoritmo de corte de aristas para vértices de grado inferior y otro para vértices de grado superior. Específicamente, dada una arista dirigida eu,v , si el grado de v es bajo, se aplica hashes a v , y si es alto, se aplica hashes a u .

Ejemplo 10.7 Considere nuevamente el grafo de la Fig. 10.22a. En la Fig. 10.23 se muestra una partición disjunta de aristas de este grafo, donde los vértices replicados se representan mediante círculos de puntos. Esta partición se logró mediante el algoritmo hash descrito anteriormente. Observe que esto provoca la replicación de 6 de los 9 vértices.

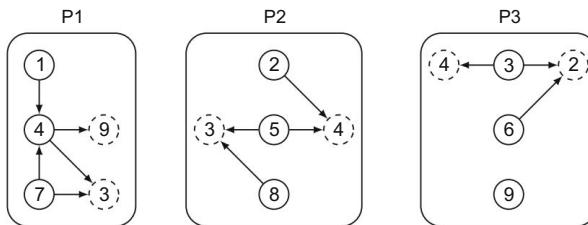


Fig. 10.23 Partición con aristas disjuntas (corte de vértice)

10.4.2 MapReduce y análisis de gráficos

Es posible utilizar un sistema MapReduce como Hadoop para el procesamiento y análisis de grafos. Sin embargo, como se mencionó en la sección 10.2.1, los sistemas MapReduce no gestionan el cálculo iterativo de forma especialmente eficaz, y en dicha sección se analizaron los principales problemas. En los sistemas de grafos, existe el problema adicional de la asignación equilibrada de vértices entre los trabajadores debido a la distribución de grados sesgada en muchos grafos reales, como se explicó en la sección 10.4.1, lo que genera variaciones en la sobrecarga de comunicación entre los nodos trabajadores. Todo esto genera una sobrecarga significativa que afecta negativamente al rendimiento de los sistemas MapReduce para el análisis de grafos. No obstante, la mayoría de los sistemas de análisis de grafos específicos que se analizarán en la siguiente sección requieren que todo el grafo se mantenga en memoria; cuando esto no es posible, MapReduce podría ser una alternativa razonable, y existen estudios que analizan su uso para diversas cargas de trabajo, así como modificaciones que permitirían una mejor escalabilidad. También existen sistemas que modifican MapReduce para adaptarse mejor a las cargas de trabajo de análisis de grafos iterativos. Como se mencionó anteriormente, Spark es una mejora de MapReduce para gestionar la iteración, y GraphX se desarrolló como un sistema de procesamiento de gráficos basado en Spark. Otra variante de MapReduce para el procesamiento de gráficos es el sistema HaLoop. Ambos separan el estado que cambia durante las iteraciones de los datos invariantes que no cambian y almacenan en caché estos últimos para evitar operaciones de E/S innecesarias. También modifican el planificador para garantizar que los mismos datos se asigne a los mismos trabajadores. Los enfoques para implementarlos difieren obviamente: HaLoop modifica el programador de tareas de Hadoop en el maestro y el rastreador de tareas en los trabajadores, e implementa un control de bucle en el maestro para verificar el punto fijo. GraphX, por otro lado, utiliza modificaciones incorporadas en Spark para gestionar mejor las cargas de trabajo iterativas. Realiza una partición disjunta de aristas del grafo y crea tablas de vértices y tablas de aristas en cada trabajador. Cada entrada en la tabla de vértices incluye el identificador de vértice junto con las propiedades del vértice, mientras que cada entrada en la tabla de aristas incluye los puntos finales de cada arista, así como las propiedades de la arista. Estas tablas se implementan como RDD de Spark. Cualquier cálculo de grafos implica un proceso de dos pasos (con iteración): unir las tablas de vértices y aristas, y realizar una agregación. La unión implica mover las tablas de vértices a los trabajadores que contienen las tablas de aristas apropiadas, ya que el número de vértices es menor que el número de aristas. Para evitar...

Al transmitir cada tabla de vértices a todos los nodos de trabajo de la tabla de aristas, GraphX crea una tabla de enrutamiento que especifica, para cada vértice, los nodos de trabajo de las tablas de aristas donde existe; esto también se implementa como un RDD.

10.4.3 Sistemas de análisis de gráficos para fines especiales

Ahora nos centraremos en los sistemas desarrollados específicamente para el análisis de grafos. Estos sistemas se pueden caracterizar según sus modelos de programación y de computación. Los modelos de programación especifican cómo un desarrollador de aplicaciones escribiría los algoritmos para su ejecución en un sistema, mientras que los modelos de computación indican cómo el sistema subyacente los ejecutaría.

Hay tres modelos de programación fundamentales: centrado en vértices y basado en particiones. centrado y centrado en el borde:

- **Modelo centrado en el vértice**

El enfoque centrado en vértices requiere que el programador se centre en el cálculo que se realizará en cada vértice. Por lo tanto, esto se conoce comúnmente como el enfoque de "pensar como un vértice". Un vértice v basa su cálculo únicamente en su propio estado y en el estado de sus vértices vecinos. Por ejemplo, en el cálculo de PageRank, cada vértice se programa para recibir el cálculo de rango de sus vecinos y calcular su propio rango con base en este. Los resultados del cálculo de estado están entonces disponibles para los vértices vecinos para que puedan realizar su cálculo.

- **Modelo centrado en particiones**

En los sistemas que siguen este modelo de programación, se espera que el programador especifique el cálculo que se realizará en una partición completa, en lugar de en cada vértice. Esto también se conoce como centrado en bloques, ya que el cálculo se realiza sobre bloques de vértices. Este enfoque también se conoce como "pensar como un bloque" o "pensar como un grafo".

El enfoque centrado en particiones suele utilizar un algoritmo serial dentro de cada bloque y solo se basa en los estados de bloques vecinos completos en lugar de estados de vértices individuales. Esta característica de utilizar algoritmos de cálculo separados dentro de una partición y entre particiones (para los vértices en los límites) puede resultar en algoritmos más complejos, pero reduce la dependencia de los estados vecinos y la sobrecarga de comunicación.

- **Modelo centrado en aristas**

Un tercer enfoque es el dual del modelo

centrado en vértices en el que

las operaciones se especifican para cada arista en lugar de cada vértice. En este caso, el objeto principal de atención es una arista en lugar de un vértice. Siguiendo el mismo esquema de nomenclatura, esto puede llamarse "pensar como una arista".

Los modelos de cálculo son paralelo síncrono masivo (BSP), paralelo asíncrono (AP) y recopilación-aplicación-dispersión (GAS):

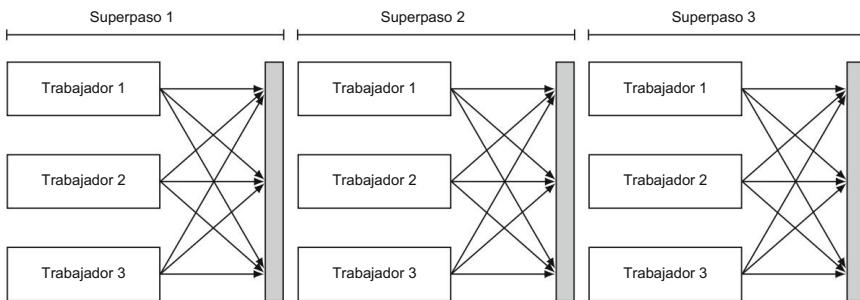


Figura 10.24 Modelo de cálculo de BSP

Paralelo Sincrónico Masivo (BSP)

es un modelo de computación paralela en el que un cálculo se divide en una serie de superpasos separados por barreras globales. En cada superpaso, todos los nodos de procesamiento (es decir, las máquinas de trabajo) realizan el cálculo en paralelo y, al final del superpaso, se sincronizan antes de comenzar el siguiente. La sincronización implica compartir el estado calculado en ese superpaso con otros para que este estado pueda ser utilizado por todos los nodos de trabajo en el siguiente. El cálculo se realiza en múltiples superpasos hasta alcanzar un punto fijo. La Figura 10.24 muestra un ejemplo de computación BSP que alcanza el punto fijo en tres superpasos que involucran tres nodos procesadores.

Dado que la mayoría de estos sistemas se ejecutan en clústeres paralelos, la comunicación suele basarse en el paso de mensajes (esto aplica a todos los modelos computacionales que analizamos). El modelo BSP implementa un enfoque de comunicación basado en push: los mensajes son enviados por el emisor y almacenados en el búfer del receptor. La recepción de un mensaje al final de un superpaso hace que el receptor se programe automáticamente para su ejecución en el siguiente. El modelo BSP simplifica la computación paralela, pero requiere un cuidadoso particionamiento de tareas para que las máquinas de trabajo estén razonablemente equilibradas y evitar rezagadas. También genera sobrecarga de sincronización al final de cada superpaso.

- Paralelo asíncrono

El modelo paralelo asíncrono (AP) elimina la restricción del modelo BSP que requiere sincronización entre máquinas de trabajo en la barrera global: los estados calculados en el superpaso k están disponibles para su uso en el cálculo del superpaso $k + 1$, incluso si llegan al destino dentro del superpaso k . El modelo AP mantiene las barreras globales para separar los superpasos, pero permite que los estados recibidos se visualicen y utilicen inmediatamente. Por lo tanto, el cálculo en el estado k puede basarse en los estados de los vecinos que se calcularon en el superpaso $k - 1$, pero que se retrasaron y no se recibieron hasta el final del superpaso $k - 1$, o en el superpaso k .

El hecho de que el cálculo del estado en un superpaso pueda solaparse con la recepción de estados de vecinos genera problemas de consistencia: los cambios y las lecturas de estados requieren un control minucioso. Esto suele gestionarse mediante la aplicación de bloques en los estados mientras se leen o escriben; dado que los estados son

Distribuidos en múltiples nodos de trabajo, es necesario contar con soluciones de bloqueo distribuido.

Un objetivo importante del modelo AP es mejorar el rendimiento al permitir que las unidades de procesamiento más rápidas continúen su procesamiento sin tener que esperar hasta que se supere la barrera de sincronización.⁸ Esto puede resultar en un menor número de superpasos. Sin embargo, dado que aún conserva las barreras de sincronización, la sobrecarga de sincronización y comunicación no se elimina por completo.

- Reunir-APLICAR-Dispersar

Como su nombre indica, el modelo de recopilación-aplicación-dispersión (GAS) consta de tres fases: en la fase de recopilación, un elemento del grafo (vértice, bloque o arista) recibe (o extrae) información sobre su vecindad; en la fase de aplicación, utiliza los datos recopilados para calcular su propio estado; y en la fase de dispersión, actualiza los estados de su vecindad. Una característica diferenciadora importante de GAS es la separación entre la actualización de estado y la activación. Tanto en BSP como en AP, cuando se comunica una actualización de estado a los vecinos, estos se activan automáticamente (es decir, se programan para su ejecución), mientras que en GAS, ambas acciones son independientes. Esta separación es importante, ya que permite al programador tomar sus propias decisiones sobre qué elementos del grafo ejecutar a continuación (quizás basándose en prioridades).

El GAS puede ser síncrono o asíncrono. El GAS síncrono es similar al modelo BSP en que se mantienen las barreras globales, pero con una diferencia importante: en BSP, cada elemento del grafo transfiere su estado a sus vecinos al final de un superpaso, mientras que en el GAS, un elemento del grafo activado transfiere el estado de sus vecinos al inicio de un superpaso.

El GAS asíncrono elimina las barreras globales y, por lo tanto, tiene el mismo problema de consistencia que el modelo AP que se aborda mediante el bloqueo distribuido.

Sin embargo, difiere del modelo AP, ya que no utiliza la noción de superpasos como el modelo BSP. Se ejecuta programando iterativamente un elemento del grafo para su ejecución, recopilando sus estados vecinos (denominados alcance), calculando su estado y actualizando el alcance y la lista de elementos del grafo que requieren programación. El cálculo finaliza cuando ya no quedan elementos del grafo pendientes de programación.

La combinación de modelos de programación y computación define el espacio de diseño con nueve alternativas. Sin embargo, hasta la fecha, no se han construido sistemas para cada una de ellas. La mayor parte de la investigación y el desarrollo se han centrado en sistemas BSP centrados en vértices (Sección 10.4.4); por lo tanto, nuestro análisis de esta clase será más profundo que el de las demás. Para los casos en los que no se conocen sistemas reales, indicamos brevemente cómo podría ser uno.

Utilizamos los términos barrera global, barrera de sincronización global y barrera de sincronización indistintamente.

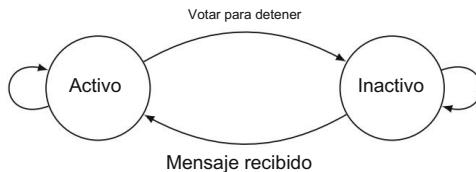


Fig. 10.25 Estados de vértice en sistemas centrados en vértices

10.4.4 Bloque síncrono centrado en el vértice

Como se mencionó anteriormente, los sistemas centrados en vértices requieren que el programador se centre en el cálculo que se realizará en cada vértice; las aristas no son objetos de primera clase en estos sistemas, ya que no se realiza ningún cálculo en ellas. Al combinarse con el modelo de cálculo BSP, estos sistemas realizan el cálculo iterativamente (es decir, en superpasos), de modo que en cada iteración, cada vértice v accede al estado contenido en los mensajes que se le enviaron en la iteración anterior, calcula su nuevo estado basándose en estos mensajes y comunica su estado a sus vecinos (quienes leerán el estado en el superpaso posterior). El sistema espera entonces hasta que todas las máquinas de trabajo completen el cálculo en esa iteración (la barrera global) antes de comenzar la siguiente iteración.

Cada vértice se encuentra en estado "activo" o "inactivo". El cálculo comienza con todos los vértices en estado activo y continúa hasta que todos alcanzan el punto fijo y entran en estado inactivo, sin que haya mensajes pendientes en el sistema (Fig. 10.25). Al alcanzar el punto fijo, cada vértice envía un mensaje de "detención de votación" antes de entrar en estado inactivo; una vez inactivo, el vértice permanece en ese estado a menos que reciba un mensaje externo para volver a activarse.

Esta categoría ha sido la más popular entre los desarrolladores de sistemas, como mencionamos anteriormente. Los sistemas clásicos son Pregel y su contraparte de código abierto, Apache Giraph. Otros son GPS, Mizan, LFGraph, Pregelix y Trinity. Nos centraremos en Pregel como ejemplo de esta clase de sistemas al analizar varios detalles (estos sistemas se conocen comúnmente como "similares a Pregel").

Para facilitar el cálculo centrado en vértices, se proporciona una función Compute() para cada vértice, y el programador debe especificar el cálculo que debe realizarse según la semántica de la aplicación. El sistema proporciona funciones integradas como GetValue() y WriteValue() para leer el estado asociado a un vértice y modificarlo, así como una función SendMsg() para enviar las actualizaciones del estado del vértice a los vértices vecinos. Estas se proporcionan como funcionalidad básica y el programador puede centrarse en el cálculo que debe realizarse en cada vértice. En este sentido, el enfoque es similar a MapReduce, donde se espera que el programador proporcione los códigos específicos para las funciones map() y reduce(), mientras que el sistema subyacente proporciona el mecanismo de ejecución y comunicación.

La función Compute() es bastante general; además de calcular un nuevo estado para el vértice, puede causar cambios en la topología del grafo (mutaciones) si el sistema lo permite. Por ejemplo, un algoritmo de agrupamiento puede reemplazar un conjunto de vértices por un solo vértice. Las mutaciones que se realizan en un superpaso son efectivas al comienzo del siguiente. Naturalmente, pueden surgir conflictos cuando varios vértices requieren la misma mutación, como la adición del mismo nodo con diferentes valores. Estos conflictos se resuelven mediante la ordenación parcial de las operaciones y la implementación de controladores definidos por el usuario. La ordenación parcial de las operaciones impone el siguiente orden: primero se eliminan las aristas, luego las eliminan de vértices, luego las adiciones de vértices y, finalmente, las adiciones de aristas. Todas estas mutaciones preceden a la llamada a la función Compute() .

Ejemplo 10.8 Para demostrar el método de cálculo BSP centrado en vértices, calcularemos las componentes conexas del grafo de la figura 10.26a. Para este ejemplo, elegimos un grafo más simple que el que usamos para la partición (figura 10.22a) para ilustrar fácilmente los pasos del cálculo.

Nótese que, dado que este grafo es dirigido, calcular la componente conexa se reduce a calcular la componente débilmente conexa (WCC), donde se ignoran las direcciones (véase el Ejemplo 10.5). Además, dado que este grafo es completamente conexo, todos los vértices deberían estar en un grupo, por lo que utilizamos este hecho para comprobar la exactitud del cálculo.

La versión BSP centrada en vértices del algoritmo WCC es la siguiente. Cada vértice guarda información sobre el grupo al que pertenece y, en cada superpaso, la comparte con sus vecinos. Al comienzo del siguiente superpaso, cada vértice obtiene estos identificadores de grupo de sus vecinos y selecciona el identificador de grupo más pequeño como su nuevo identificador de grupo: Compute() = min{identificadores de grupo de vecinos, identificador de grupo propio}. Si su identificador de grupo no ha cambiado desde el superpaso anterior, el vértice entra en estado inactivo (recuerde que el estado inactivo representa que el valor del vértice ha alcanzado el punto fijo). De lo contrario, envía su nuevo identificador de grupo a sus vecinos. Cuando un vértice entra en estado inactivo, no envía más mensajes a sus vecinos, pero recibirá mensajes de sus vecinos activos para determinar si debe volver a activarse. El cálculo continúa de esta manera a lo largo de múltiples superpasos.

Esta ejecución se muestra en la Fig. 10.26b. En el paso de inicialización, el algoritmo se inicializa asignando cada vértice a su propio grupo, identificado por su id de vértice (p. ej., el vértice v1 pertenece al grupo 1). El valor de cada vértice es el estado al final del superconjunto etiquetado. A continuación, este id de grupo se envía a sus vecinos. Cada flecha muestra cuándo el trabajador receptor consume un mensaje; por lo tanto, apuntar al siguiente superconjunto significa que no se accede al mensaje hasta entonces, independientemente de cuándo se entregue o reciba. En el superpaso 1, observe que los vértices v4, v7, v5, v8, v6 y v9 cambian sus id de grupo, mientras que los vértices v1, v2 y v3 no cambian sus valores y entran en estado inactivo. El cálculo completo consta de 9 superpasos en este ejemplo.

Observe que, en algunos casos, los vértices inactivos se activan como resultado de los mensajes que reciben de sus vecinos. Por ejemplo, el vértice v2 , que se inactiva en el superpaso 2, se activa en el superpaso 4 al recibir un ID de grupo 1 de v7 , lo que provoca que actualice su propio ID de grupo. Esta es una característica de este tipo de cálculo, como se mencionó anteriormente.

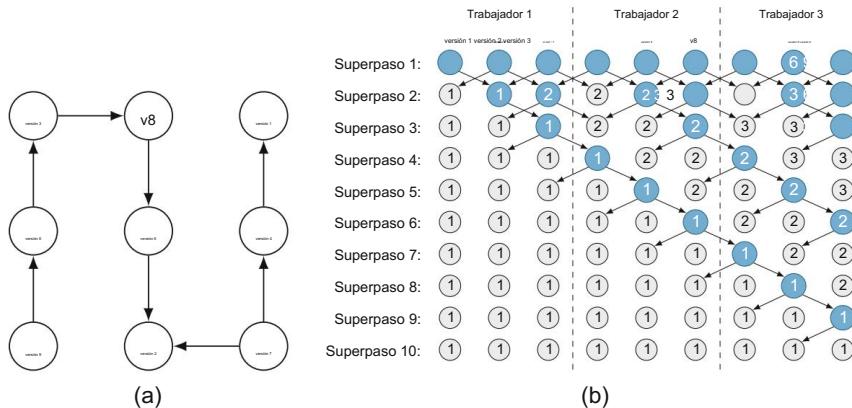


Fig. 10.26 Ejemplo de BSP centrado en el vértice. (a) Ejemplo de gráfico. (b) Cálculo de BSP centrado en el vértice de WCC (los vértices grises están inactivos, los vértices azules están activos)

Recuerde que estos sistemas realizan cálculos paralelos sobre un clúster donde hay un nodo maestro y una serie de nodos de trabajo, y cada trabajador aloja un conjunto de vértices de grafos e implementación de la función Compute(). En algunos sistemas (por ejemplo, GPS y Giraph), hay una función Master.Compute() adicional que permite para que algunas partes del algoritmo se ejecuten en serie en el maestro. La existencia de estas funciones proporcionan mayor flexibilidad para la implementación del algoritmo y algunas optimizaciones (como discutiremos más adelante).

Para algunos algoritmos, es importante capturar el estado global del gráfico.

Para facilitar esto, se puede implementar un agregador. Cada vértice aporta un valor a el agregador, y el resultado de la agregación se pone a disposición de todos los vértices. En el siguiente superpaso. Los sistemas suelen proporcionar varios agregadores básicos, como mínimo, máximo y suma.

El rendimiento de los sistemas de esta categoría se ve afectado por dos factores: el coste de la comunicación y el número de superpasos. Las propiedades de los grafos reales que...

Hemos analizado anteriormente el impacto de los dos factores de coste mencionados anteriormente:

1. Gráficos de ley de potencia con distribución de grados sesgada: El problema con el grado

La asimetría de la distribución es que los trabajadores que tienen estos vértices de alto grado reciben y tienen que procesar muchos más mensajes que otros, lo que genera desequilibrios de carga entre los trabajadores que causan el problema del rezagado que analizamos anteriormente.

2. Alto grado de vértice promedio: Esto da como resultado que cada vértice tenga que lidiar con un alto grado de vértice promedio.

Número de mensajes entrantes y tener que comunicarse con un gran número de vértices vecinos, lo que genera una gran sobrecarga de comunicación.

3. Diámetros grandes: Si en el cálculo del BSP cada superpaso corresponde a un

salto (es decir, un mensaje) entre vértices, estos cálculos tomarán una gran cantidad de superpasos, proporcional al diámetro del grafo. Aunque un unos pocos cientos de saltos como diámetro de gráfico pueden no parecer excesivos, tenga en cuenta que muchas de las cargas de trabajo de análisis requieren múltiples pasadas sobre todos los vértices en

Estos grafos, lo que genera un alto costo algorítmico. Se ha informado, por ejemplo, que ejecutar un algoritmo de componentes fuertemente conectados en un grafo de diámetro 20 requiere más de 4500 superpasos (sin optimización).

Una optimización del sistema que puede implementarse para reducir la sobrecarga de comunicación entre nodos trabajadores es un combinador que combina los mensajes destinados al vértice v según la semántica definida por la aplicación (p. ej., si v solo necesita la suma de los valores de los vecinos). Esto no puede realizarse automáticamente, ya que el sistema no puede determinar cuándo y cómo es apropiada esta agregación; en su lugar, el sistema proporciona una función `Combine()` cuyo código debe especificar el programador.

Las optimizaciones a nivel de sistema para abordar la asimetría incluyen la implementación de algoritmos de partición de grafos sensibles a esta. Los sistemas basados en particiones que analizamos en las Secciones 10.4.7–10.4.9 también abordan estos problemas mediante un diseño de sistema radicalmente diferente.

También se han propuesto optimizaciones algorítmicas para abordar estos problemas, pero estas requieren modificaciones en la implementación de los algoritmos de carga de trabajo. Aunque no las analizaremos en detalle, destacamos una como ejemplo. Algunos algoritmos de carga de trabajo analítica pueden provocar que una pequeña porción de los vértices del grafo permanezca activa después de que los demás se vuelvan inactivos, lo que genera muchos más superpasos antes de la convergencia. En esta optimización, si la parte "activa" del grafo es suficientemente pequeña, el cálculo se traslada al nodo maestro y se ejecuta en serie mediante la función `Master.Compute()`. Se ha demostrado experimentalmente que esto puede reducir el número de superpasos entre un 20 % y un 60 %.

10.4.5 Asíncrono centrado en vértices

Estos sistemas siguen el mismo modelo de programación que el caso anterior, pero relajan el modelo de ejecución síncrona, manteniendo al mismo tiempo las barreras de sincronización al final de cada superpaso. En consecuencia, la función `Compute()` de cada vértice se ejecuta en cada superpaso, como se indicó anteriormente, y los resultados se envían a los vértices vecinos. Sin embargo, los mensajes disponibles como entrada para la función no se limitan a los enviados en el superpaso anterior; un vértice puede ver los mensajes que recibe dentro del mismo superpaso en el que se envió. Los mensajes que no están disponibles al ejecutar `Compute()` se recogen al principio del superpaso siguiente, como en BSP. Este enfoque soluciona un problema importante del modelo BSP, a la vez que mantiene la facilidad de la programación centrada en vértices: un vértice puede ver mensajes más recientes que no se retrasan hasta el superpaso siguiente. Esto suele dar como resultado una convergencia más rápida que en los sistemas basados en BSP. GRACE y GiraphUC siguen este enfoque.

Ejemplo 10.9 Para demostrar sistemas AP centrados en vértices, utilizamos el ejemplo del componente débilmente conectado de la sección anterior (Ejemplo 10.8).

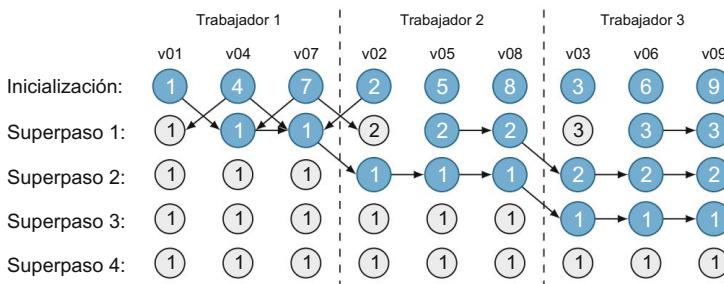


Fig. 10.27 Ejemplo de AP centrado en el vértice

Para simplificar las cosas, asumimos que todos los mensajes entre vértices llegan a sus destinos dentro del mismo superpaso, y que el cálculo en cada trabajador es también se completa en el mismo superpaso. Además, asumimos un solo subproceso ejecución donde cada trabajador ejecuta Compute() en los vértices uno a la vez.

Bajo estos supuestos, el cálculo del WCC sobre el gráfico de la Fig. 10.26a se muestra en la Fig. 10.27. Nótese, por ejemplo, que durante el paso de inicialización, v1 empuja su id de grupo (1) a v4, y v4 empuja su id de grupo (4) a v1 y v7. Dado que Supongamos una ejecución de un solo subproceso; se ejecuta v4.Compute para cambiar el grupo de v4. id a 1, que también se envía a v7 en el mismo superpaso (superconjunto 1). Entonces, cuando Se ejecuta v7.Compute(), el id de grupo de v7 se establece en 1. El acceso a los estados del vértice Dentro del mismo superpaso está la característica distintiva del modelo AP.

Como se mencionó anteriormente, la ejecución asíncrona requiere control de consistencia mediante bloqueo distribuido. Esto se manifiesta en esta clase de sistemas de la siguiente manera. El estado de un vértice puede ser accedido por otros vértices mientras ejecutan su Funciones Compute(): es posible que el vértice vi esté ejecutando su función Compute() mientras su El vértice vecino vj envía sus actualizaciones a vi. Para evitarlo, se establecen bloqueos. en cada vértice. Dado que los vértices se distribuyen entre los nodos de trabajo, esto requiere un mecanismo de bloqueo distribuido para garantizar la consistencia del estado.

Otro problema con el enfoque AP es que rompe un tipo diferente de consistencia. Garantía de ejecución proporcionada por BSP. Cada vértice ejecuta Compute() con todos el mensaje que ha recibido desde la última ejecución, y este será una mezcla de mensajes antiguos (del superpaso anterior) y mensajes nuevos (del actual) superpaso). Por lo tanto, no es posible argumentar que, en cada superpaso, cada vértice Calcula consistentemente un nuevo estado basado en los estados de los vértices vecinos en el final del superpaso anterior. Sin embargo, esta relajación permite que un vértice se ejecute su función Compute() tan pronto como recibe un mensaje de alta prioridad, por ejemplo.

Aunque el modelo AP aborda el problema de obsolescencia de los mensajes de BSP, aún... tiene cuellos de botella en el rendimiento debido a la existencia de barreras de sincronización global, Es decir, tiene una alta sobrecarga de comunicación y tiene que lidiar con rezagados. Estos Se puede superar eliminando algunas de las barreras como se propone en el modelo sin barreras. Modelo paralelo asíncrono (BAP) de GiraphUC. BAP mantiene barreras globales, pero

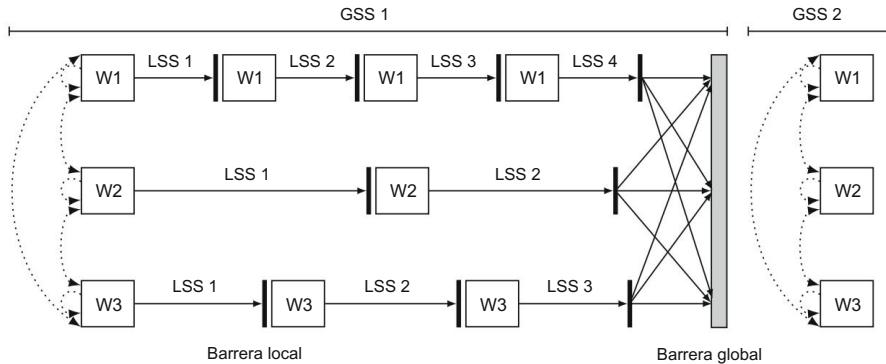


Fig. 10.28 Modelo BAP sobre tres nodos de trabajo

Divide cada superpaso global en superpasos lógicos separados por barreras locales muy ligeras. Esto permite que los trabajadores rápidos ejecuten la función Compute() varias veces (una vez en cada superpaso lógico) antes de que necesite sincronizarse globalmente con los nodos trabajadores más lentos. Al igual que en AP, los vértices pueden leer inmediatamente los mensajes locales y remotos que han recibido, lo que reduce la obsolescencia de los mensajes. La Figura 10.28 muestra BAP sobre tres nodos trabajadores; el primer trabajador (W1) tiene cuatro superpasos lógicos (LSS) dentro del primer superpaso global (GSS), el segundo trabajador tiene dos y el tercero tiene tres. Las flechas punteadas representan los mensajes recibidos y procesados en el mismo superpaso lógico, mientras que las flechas sólidas son los mensajes que se recogen en el siguiente superpaso global.

El modelo BAP requiere cuidado al determinar la terminación. Recuerde que, tanto en los modelos BSP como AP, la terminación se verifica en cada barrera de sincronización comprobando que todos los vértices estén inactivos y que no haya mensajes en transmisión. Dado que ahora existe una separación entre las barreras locales y globales, esta verificación debe realizarse tanto en la barrera local como en la global. Un enfoque sencillo consiste en verificar, en una barrera local, si hay más mensajes locales o remotos que procesar; de no ser así, no hay más trabajo que realizar y los vértices de este nodo de trabajo llegan a la barrera global. Cuando todos los vértices del grafo llegan a la barrera global, se realiza una segunda verificación, similar a la de BSP y AP: el cálculo finaliza globalmente si todos los vértices están inactivos y no hay más mensajes.

10.4.6 Recopilar-Aplicar-Dispersión Centrado en el Vértice

Esta categoría se caracteriza por GraphLab, que combina el modelo de programación centrado en vértices con el modelo de computación GAS basado en pull. Como se mencionó anteriormente, puede existir una versión sincrónica de este enfoque (como la implementada en GraphLab Sync) que es prácticamente igual a la BSP centrada en vértices (excepto el aspecto pull), por lo que...

No se profundizará en este tema. La versión asíncrona es diferente: en la fase de recopilación, un vértice extrae datos de sus vecinos en lugar de que estos envíen sus datos.⁹ Para cada vértice va, se define un ámbito [Scope(v)] que consiste en los datos almacenados en todas las aristas y vértices adyacentes, así como los datos en v. La función Compute() (en GraphLab, llamada función Update) toma como entrada v y Scope(v) y devuelve el Scope(v) actualizado, así como un conjunto de vértices V cuyos estados han cambiado y son candidatos para la programación. La ejecución sigue tres pasos, tomando como entrada un grafo G y un conjunto inicial de vértices V:

1. Elimine un vértice de V según la decisión de programación.
2. Ejecute la función Compute() y calcule Scope (v) y V.
3. $V \leftarrow V - V'$.

Estos tres pasos se ejecutan iterativamente hasta que no haya más vértices en V. La separación de las actualizaciones de estado en el Ámbito (v) (es decir, los estados de los vecinos) de la programación de los cálculos de vértices es una distinción importante entre GAS y el enfoque AP, donde los mensajes que actualizan los estados de los vértices también programan dichos vértices para su cálculo. Esta separación permite flexibilidad para elegir el orden de los cálculos de vértices, por ejemplo, según prioridades o equilibrio de carga. Además, cabe destacar que no existe una función SendMsg() explícita en la ejecución de GAS; la compartición de los cambios de estado se realiza en la fase de recopilación.

Dado que un vértice v puede leer datos directamente de su Scope(v), pueden surgir inconsistencias, ya que múltiples cálculos de vértices pueden causar actualizaciones de estado conflictivas, como se mencionó anteriormente, lo que requiere la implementación de mecanismos de bloqueo distribuido. Cuando el vértice v ejecuta Compute, obtiene bloqueos sobre su Scope(v), realiza su cálculo, actualiza Scope(v) y luego libera sus bloqueos. En GraphLab, esto se conoce como consistencia total. En ese sistema en particular, se proporcionan dos niveles de consistencia más flexibles para adaptarse mejor a las aplicaciones cuya semántica no requiere exclusión mutua total: consistencia de aristas y consistencia de vértices. La consistencia de aristas garantiza que v tenga acceso de lectura/escritura a sus propios datos y a los datos de sus aristas adyacentes, pero solo acceso de lectura a sus vértices vecinos. Por ejemplo, el cálculo de PageRank solo requeriría consistencia de aristas, ya que solo lee los rangos de los vértices vecinos. La consistencia de vértices simplemente garantiza que mientras v ejecuta su función Compute(), ningún otro vértice accederá a él. La consistencia de los bordes y vértices permite tener en cuenta la semántica de la aplicación para lograr la coherencia.

10.4.7 Procesamiento síncrono de bloques centrado en particiones

Como señalamos en la Sección 10.4.4, muchos gráficos de la vida real presentan propiedades que son desafiantes para los sistemas centrados en vértices, y se han realizado varias optimizaciones.

⁹GraphLab también se diferencia por su implementación de memoria compartida distribuida, pero eso no es importante en esta discusión.

Desarrollados para abordar los problemas planteados, los sistemas BSP centrados en particiones constituyen un enfoque diferente para abordar estos problemas. Estos sistemas aprovechan la partición del grafo en nodos trabajadores, de modo que, en lugar de que cada vértice se comunique con los demás mediante el paso de mensajes, como en el enfoque centrado en vértices, la comunicación se limita a mensajes entre bloques (particiones) con un algoritmo serial más simple implementado dentro de cada partición. El cálculo sigue el BSP, por lo que se realizan múltiples iteraciones como superpasos hasta que el sistema converge. Este enfoque se ejemplifica con Blogel y Giraph++.

El punto crítico de estos sistemas es que ejecutan un algoritmo serial dentro de bloques y solo se comunican entre bloques. Una forma de razonar sobre esto es que dado un grafo $G = (V, E)$, después de la partición, tenemos un grafo $G = (B, E)$, donde B es el conjunto de bloques y E es el conjunto de aristas entre bloques. En algoritmos centrados en particiones, la sobrecarga de comunicación está limitada por $|E|$, que es significativamente menor que $|E|$; por lo tanto, los grafos con alta densidad tienen baja sobrecarga de comunicación. Por ejemplo, un experimento realizado en el grafo de Friendster para calcular componentes conexos muestra que un sistema centrado en vértices requiere 372 veces más mensajes y un tiempo de cálculo 48 veces mayor que un sistema centrado en particiones.

Los sistemas basados en particiones también reducen el diámetro de los grafos, ya que cada bloque está representado por un vértice en G , lo que resulta en una reducción significativa del número de superpasos en el cálculo de BSP. Un experimento similar para calcular componentes conectados en el grafo de la red de carreteras de EE. UU. (que tiene un diámetro aproximado de 9000) demuestra que el número de superpasos se reduce de más de 6000 a 22.

Finalmente, el manejo de la asimetría en la distribución de grados se realiza mediante el algoritmo de partición de grafos, que garantiza un número equilibrado de vértices en cada bloque. Dado que se ejecuta un algoritmo serial dentro de cada bloque, los vértices de mayor grado no necesariamente generan un gran número de mensajes; nuevamente, el argumento es que los sistemas centrados en vértices funcionan en G , mientras que los sistemas centrados en particiones funcionan en G , significativamente menor.

Ejemplo 10.10 Consideremos cómo se realizaría el cálculo WCC que hemos estado analizando en un sistema BSP centrado en particiones. Los pasos del cálculo se muestran en la Fig. 10.29, donde el segmento del grafo en cada trabajador es una partición denotada por sombreado. Dado que se utiliza un algoritmo serial dentro de cada partición, el algoritmo que hemos estado analizando, es decir, dónde comienza cada vértice en su propio grupo, no es necesariamente el que se podría utilizar, pero para comparar con enfoques anteriores, asumiremos el mismo algoritmo. En el superpaso 1, cada nodo trabajador realiza un cálculo serial para determinar los identificadores de grupo de los vértices en su partición. Para el trabajador 1, el identificador de grupo más pequeño es 1, por lo que se convierte en el identificador de grupo de los vértices $v1$, $v4$ y $v7$. Un cálculo similar se realiza en otros trabajadores. Al final del superpaso, cada trabajador transfiere su identificador de grupo a los demás trabajadores y el cálculo se repite.

Obsérvese que el número de superpasos en este caso es el mismo que el del AP centrado en vértices (Ejemplo 10.9); en general, podría ser menor, pero este no es el punto principal. El ahorro radica en el número de mensajes que se intercambian: el enfoque centrado en particiones intercambia solo 6 mensajes, mientras que el AP centrado en vértices intercambia 20.

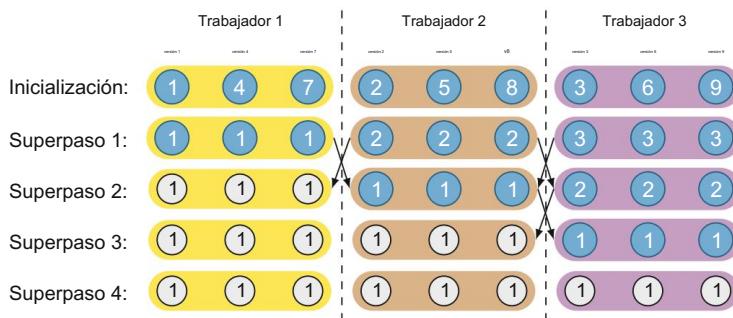


Fig. 10.29 Ejemplo de BSP centrado en partición

Los gráficos de ejemplo que utilizamos en estos ejemplos no están densamente conectados; si lo estuvieran, los ahorros habrían sido más sustanciales.

10.4.8 Asíncrono centrado en particiones

Los sistemas de esta categoría dividirían el gráfico entre nodos de trabajo como en Secc. 10.4.7 y ejecutar un algoritmo serial dentro de cada partición, pero comunicarse entre trabajadores de forma asíncrona cuando se envían mensajes entre particiones. Como se indicó anteriormente, la comunicación asíncrona generalmente se implementaba mediante un sistema distribuido. Bloqueo. En ese sentido, estos sistemas serían muy similares a los SGBD distribuidos. que hemos estado discutiendo en este libro, si uno trata cada partición de trabajador como una Fragmento de datos. En este momento, no se han desarrollado sistemas. en esta categoría.

10.4.9 Recopilar-APLICAR-Dispersión Centrado en Particiones

La única diferencia en este caso con el BSP centrado en la partición descrito en la Sección 10.4.7 es el uso de GAS basado en pull en lugar de BSP basado en push. Nuevamente, esta clase de Los sistemas son muy similares a los DBMS distribuidos con cambios apropiados en los datos. operaciones de transferencia en los planes de consulta. Hasta el momento, no hay sistemas que hayan sido... desarrollado en esta categoría.

10.4.10 Procesamiento síncrono de bloques centrado en el borde

Los sistemas centrados en aristas se centran en cada arista como objeto principal de interés; en este sentido, son duales de los enfoques centrados en vértices. El cálculo se realiza en cada arista y se itera mediante superpasos hasta alcanzar el punto fijo. Sin embargo, cabe destacar que una arista en un grafo se identifica con sus dos vértices incidentes. Por lo tanto, ejecutar Compute() en una arista requiere ejecutarse en los vértices incidentes de dicha arista. La verdadera diferencia, por lo tanto, radica en que el sistema gestiona una arista a la vez, en lugar de un vértice a la vez, como en el enfoque centrado en vértices.

Una pregunta natural es por qué esto sería preferible, dado que la mayoría de los grafos tienen muchas más aristas que vértices. A primera vista, podría parecer que un enfoque centrado en aristas generaría más computación. Sin embargo, recuerde que un mayor número de aristas resulta en un alto costo de mensajería en sistemas centrados en vértices. Además, en sistemas centrados en vértices, las aristas se suelen ordenar por su vértice de origen y se construye un índice sobre ellas para facilitar el acceso. Cuando las actualizaciones de estado se propagan a los vértices vecinos, se accede aleatoriamente a este índice para localizar las aristas incidentes a ese vértice; este acceso aleatorio es costoso. Los sistemas centrados en aristas buscan contrarrestar estos problemas operando en una secuencia de aristas sin ordenar, donde cada arista identifica sus vértices de origen y destino; no hay acceso aleatorio a un índice y cuando se envían mensajes después del cálculo en una arista, solo hay un vértice de destino. Dado que estamos considerando el modelo de cálculo BSP, las actualizaciones que se calculan en un superpaso están disponibles al inicio del superpaso posterior. X-Stream sigue este enfoque en un sistema paralelo de memoria compartida e implementa optimizaciones para el procesamiento de gráficos tanto en memoria como en disco.

10.4.11 Asíncrono centrado en el borde

Los sistemas de esta categoría serían modificaciones de los sistemas asíncronos centrados en vértices (Sección 10.4.5), donde la prioridad es la ejecución consistente en cada arista, no en cada vértice. Por lo tanto, los bloqueos se implementarían en las aristas, no en los vértices. Actualmente, no se conocen sistemas de esta categoría.

10.4.12 Recopilar-Aplicar-Dispersión Centrado en el Borde

La combinación de la programación centrada en aristas con el modelo de computación GAS implicaría cambios en los sistemas BSP centrados en aristas (Sección 10.4.10), de la misma manera que los sistemas GAS centrados en vértices modifican los sistemas BSP centrados en vértices. Esto implicaría que las funciones de recopilación, aplicación y dispersión se implementan en aristas, con lectura de estado basada en extracción durante la fase de recopilación. Actualmente, no se conocen sistemas que sigan este enfoque.

10.5 Lagos de datos

Las tecnologías de big data permiten el almacenamiento y análisis de muchos tipos diferentes de datos, que pueden ser estructurados, semiestructurados o no estructurados, en su formato natural. Esto proporciona la oportunidad de abordar de una manera nueva el viejo problema de la integración física de datos (ver Cap. 7, que típicamente se ha resuelto utilizando almacenes de datos). Un almacén de datos integra datos de diferentes fuentes de datos de empresas utilizando un formato común, generalmente el modelo relacional, que requiere transformación de datos. Por el contrario, un lago de datos almacena datos en su formato nativo, utilizando un almacén de big data, como Hadoop HDFS. Cada elemento de datos se puede almacenar directamente, con un identificador único y metadatos asociados (p. ej., fuente de datos, formato, etc.). Por lo tanto, no hay necesidad de transformación de datos. La promesa es que, para cada pregunta comercial, uno puede encontrar rápidamente el conjunto de datos relevante para analizarlo.

El término data lake se introdujo para contrastar con los almacenes de datos y los data marts. Suele asociarse con el ecosistema de software Hadoop. Se ha convertido en un tema candente, aunque controvertido, sobre todo en comparación con el almacén de datos.

En el resto de esta sección, analizaremos con más detalle la diferencia entre data lake y data warehouse. A continuación, presentaremos los principios y la arquitectura de un data lake. Finalmente, abordaremos las cuestiones pendientes.

10.5.1 Lago de datos versus almacén de datos

Como se explicó en el capítulo 7, un almacén de datos se integra con bases de datos físicas. Es fundamental para las aplicaciones OLAP y de análisis de negocio. Por lo tanto, generalmente es el núcleo de la estrategia orientada a datos de una empresa. Cuando se inició el almacenamiento de datos (en la década de 1980), estos datos empresariales se almacenaban en bases de datos operativas OLTP. Hoy en día, cada vez más datos útiles provienen de muchas otras fuentes de big data, como registros web, redes sociales y correos electrónicos. Como resultado, el almacén de datos tradicional presenta varios problemas:

- Largo proceso de desarrollo. El desarrollo de un almacén de datos suele ser un proceso largo y de varios años. La razón principal es que requiere una definición y un modelado precisos de los datos necesarios. Una vez encontrados los datos necesarios, a menudo en silos de información empresarial, es necesario definir cuidadosamente un esquema global y los metadatos asociados, así como diseñar procedimientos de limpieza y transformación de datos.
- Esquema al escribir. Un almacén de datos generalmente se basa en un SGBD relacional para gestionar los datos, que se estructuran según un esquema relacional. Los SGBD relacionales adoptan lo que recientemente se ha denominado un enfoque de esquema al escribir, en contraste con el de esquema al leer (que se analizará en breve). Con el esquema al escribir, los datos se escriben en la base de datos con un formato fijo, según lo definido por el esquema. Esto ayuda a garantizar la consistencia de la base de datos. Posteriormente, los usuarios pueden formular consultas basadas en...

esquema para recuperar los datos, que ya están en el formato correcto. El procesamiento de consultas es eficiente en este caso, ya que no hay necesidad de analizar los datos en tiempo de ejecución. Sin embargo, esto es a expensas de la evolución difícil y costosa para adaptarse a los cambios en el entorno empresarial. Por ejemplo, la introducción de nuevos datos puede implicar modificaciones del esquema (p. ej., agregar nuevas columnas), lo que a su vez puede implicar cambios en las aplicaciones y consultas.

- Procesamiento de datos de carga de trabajo OLAP. Un almacén de datos generalmente está optimizado solo para cargas de trabajo OLAP, donde los analistas de datos pueden consultar los datos de forma interactiva en diferentes dimensiones, p. ej., a través de cubos de datos. La mayoría de las aplicaciones OLAP, como el análisis de tendencias y la previsión, necesitan analizar datos históricos y no necesitan las versiones más actuales de los datos. Sin embargo, las aplicaciones OLAP más recientes pueden necesitar acceso en tiempo real a los datos operativos, lo cual es difícil de soportar en un almacén de datos.
- Desarrollo complejo con ETL. La integración de fuentes de datos heterogéneas mediante un esquema global requiere el desarrollo de programas ETL complejos para gestionar la limpieza, la transformación y la actualización de datos. Con la creciente diversidad de fuentes de datos que integrar, el desarrollo ETL se vuelve aún más difícil.

Un lago de datos es un repositorio central de todos los datos empresariales en su formato natural. Al igual que un almacén de datos, puede utilizarse para aplicaciones OLAP y de análisis de negocio, así como para el análisis de datos por lotes o en tiempo real mediante tecnologías de big data. En comparación con un almacén de datos, un lago de datos ofrece las siguientes ventajas.

Esquema en lectura. Esquema en lectura se refiere al enfoque de "cargar primero" para el análisis de big data, como lo exemplifica Hadoop. Con esquema en lectura, los datos se cargan tal cual, en su formato nativo, por ejemplo, en Hadoop HDFS. Luego, al leer los datos, se aplica un esquema para identificar los campos de datos de interés. De esta manera, los datos se pueden consultar en su formato nativo. Esto proporciona mucha flexibilidad, ya que se pueden agregar nuevos datos en cualquier momento al data lake. Sin embargo, se requiere más trabajo para escribir el código que aplica el esquema a los datos, por ejemplo, como parte de la función Map en MapReduce. Además, el análisis de datos se realiza durante la ejecución de la consulta. Procesamiento de datos con múltiples cargas de trabajo. La pila de software de gestión de big data (véase la Fig. 10.1) admite múltiples métodos de acceso a los mismos datos, por ejemplo, análisis por lotes con un marco como MapReduce, OLAP interactivo o análisis de negocios con un marco como Spark, o análisis en tiempo real con un marco de transmisión de datos. Por lo tanto, al integrar estos diferentes marcos, un lago de datos puede soportar el procesamiento de datos con múltiples cargas de trabajo.

- **Arquitectura rentable.** Al basarse en tecnologías de código abierto para implementar la pila de software de gestión de big data y clústeres sin recursos compartidos, un lago de datos ofrece una excelente relación calidad-precio y un excelente retorno de la inversión.

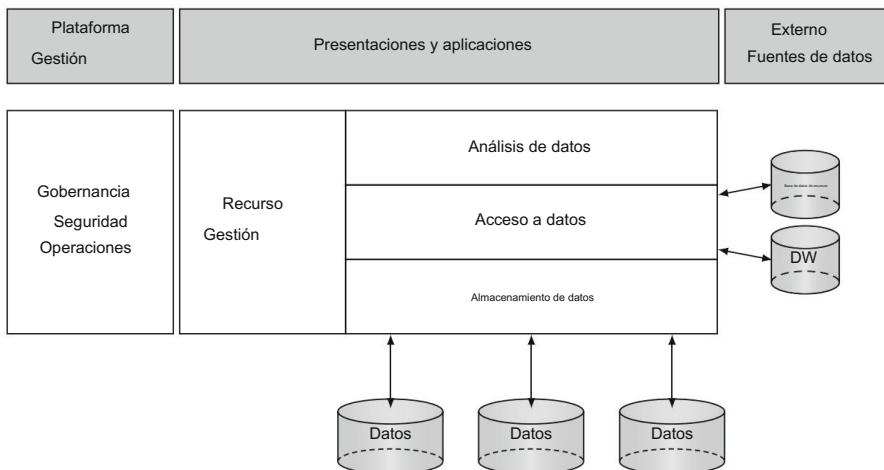


Figura 10.30 Arquitectura del lago de datos

10.5.2 Arquitectura

Un lago de datos debe proporcionar las siguientes capacidades principales:

- Recopilar todos los datos útiles: datos brutos, datos transformados, datos procedentes de fuentes de datos externas, etc.; • Permitir a los usuarios de diferentes unidades de negocio explorar los datos y enriquecerlos con metadatos;
- Acceder a datos compartidos a través de diferentes métodos: por lotes, interactivo, en tiempo real, etc. • Gobernar, proteger y gestionar datos y tareas.

La figura 10.30 muestra la arquitectura del lago de datos, con sus componentes principales.

En el centro de la arquitectura se encuentran los componentes de gestión de big data (almacenamiento, acceso, análisis y gestión de recursos), sobre los cuales se pueden crear diferentes presentaciones y aplicaciones. Estos componentes pertenecen a la pila de software de gestión de big data y se encuentran en el software de código abierto Apache. Cabe destacar que muchas herramientas de BI ya están disponibles para trabajar con Hadoop, donde podemos distinguir nuevas herramientas o extensiones de herramientas de BI tradicionales para sistemas de gestión de bases de datos relacionales (RDBMS). También podemos distinguir dos enfoques (que pueden combinarse en una sola herramienta):

1. SQL en Hadoop, es decir, utilizando un controlador SQL de Hadoop como HiveQL o Spark SQL. Ejemplos de herramientas son Tableau, Platform, Pentaho, Power BI y DB2 BigSQL.
2. Biblioteca de funciones que proporciona acceso a HDFS mediante operadores de alto nivel. Ejemplos de herramientas son Datameer, Power BI y DB2 BigSQL.

En el lado izquierdo de la arquitectura se encuentra la gestión de la plataforma, que incluye la gobernanza de datos, la seguridad de los datos y las operaciones de tareas. Estos componentes complementan

gestión de big data con funciones críticas para compartir datos a escala empresarial (entre múltiples unidades de negocio). La gobernanza de datos tiene una importancia creciente en un data lake, ya que es necesario gestionar los datos de acuerdo con la política de la empresa, con especial atención a las leyes de privacidad de datos, p. ej., el famoso Reglamento General de Protección de Datos (RGPD) adoptado por la Unión Europea en mayo de 2018. Esta política suele ser supervisada por un comité de gobernanza de datos e implementada por administradores de datos, que están a cargo de organizar los datos para las necesidades empresariales. La seguridad de los datos incluye la autenticación de usuarios, el control de acceso y la protección de datos. Las operaciones de tareas incluyen el aprovisionamiento, la monitorización y la programación de tareas (normalmente en un clúster de SN). Al igual que para las herramientas de BI para big data, ahora se pueden encontrar herramientas Apache para la gobernanza de datos, p. ej., Falcon, la seguridad de datos, p. ej., Ranger y Sentry, y las operaciones de tareas, p. ej., Ambari y Zookeeper.

Finalmente, el lado derecho de la arquitectura muestra que se pueden integrar diferentes tipos de fuentes de datos externas, por ejemplo, SQL, NoSQL, etc., normalmente utilizando los envoltorios de las herramientas para el acceso a los datos, por ejemplo, los conectores Spark.

10.5.3 Desafíos

Construir y operar un data lake sigue siendo un desafío, tanto por razones metodológicas como técnicas. La metodología para un almacén de datos ya se comprende bien.

Consiste en una combinación de modelado de datos prescriptivo (esquema en escritura), gestión de metadatos y gobernanza de datos, que en conjunto generan una sólida consistencia de los datos. Posteriormente, con potentes herramientas OLAP o de análisis de negocios, diferentes usuarios, incluso con conocimientos limitados de análisis de datos, pueden extraer valor de los datos. En particular, un data mart facilita el análisis de datos específicos para cada necesidad empresarial.

Por el contrario, un lago de datos carece de consistencia, lo que dificulta considerablemente el análisis de datos a escala empresarial. Esta es la principal razón por la que se necesitan científicos de datos y administradores de datos cualificados. Otra razón es que el panorama tecnológico del big data es complejo y está en constante evolución. Por lo tanto, se deben considerar la siguiente metodología y las mejores prácticas para construir un lago de datos:

- Establecer una lista de prioridades y valores añadidos para el negocio, en comparación con el almacén de datos de la empresa. Esto debe incluir la definición de objetivos de negocio precisos y los requisitos de datos correspondientes para el lago de datos. • Tener una visión global de la arquitectura del lago de datos, que debe ser extensible (para adaptarse a las evoluciones técnicas) e incluir la gobernanza de datos y la gestión de metadatos.
- Definir una política de seguridad y privacidad, que es fundamental si se comparten datos entre líneas de negocio.
- Definir un modelo de computación/almacenamiento que respalde la visión global. En particular, los aspectos de extensibilidad y escalabilidad determinarán las decisiones técnicas. • Definir un plan operativo con acuerdos de nivel de servicio (SLA) en términos de tiempo de actividad, volumen, variedad, velocidad y veracidad.

Las razones técnicas que dificultan un lago de datos se deben a problemas de integración y calidad de los datos. La integración de datos tradicional (véase el capítulo 7) se centra en el problema de la integración de esquemas, incluyendo la correspondencia y el mapeo de esquemas, para producir un esquema global. En el contexto de la integración de big data, el problema de la integración de esquemas, con muchas fuentes de datos heterogéneas, se vuelve más difícil. El enfoque del lago de datos simplemente evita el problema de la integración de esquemas mediante la gestión de datos sin esquema. Sin embargo, a medida que los lagos de datos maduran, puede surgir el problema de la integración de esquemas para mejorar la consistencia de los datos. Entonces, una solución interesante es la extracción automática de metadatos e información de esquemas de muchos elementos de datos relacionados, por ejemplo, dentro del mismo conjunto de datos. Una forma de lograr esto es mediante la combinación de técnicas de aprendizaje automático, correspondencia y agrupamiento.

10.6 Conclusión

El big data y su rol en la ciencia de datos se han convertido en temas importantes para la gestión de datos, aunque su definición precisa es compleja. Por consiguiente, no existe un marco unificado que permita presentar los avances en esta área; quizás la arquitectura de referencia que proporcionamos en la Fig. 10.1 sea la mejor opción. Por lo tanto, en este capítulo nos centramos en las propiedades que caracterizan a estos sistemas y en las plataformas fundamentales propuestas para abordarlos. En resumen, analizamos los sistemas de almacenamiento distribuido y las plataformas de procesamiento MapReduce y Spark que abordan los problemas de la gestión y el procesamiento de grandes volúmenes de datos; analizamos los flujos de datos que gestionan la propiedad de velocidad asociada a las aplicaciones de big data; y analizamos el análisis de grafos que, junto con los flujos de datos, destaca los problemas de variedad. El análisis de los data lakes aborda los problemas de variedad y escala en términos de integración de datos, y destaca los problemas de calidad de los datos (veracidad) y la necesidad de depuración de los datos cuando la información de origen no está bien seleccionada.

Es inevitable que los temas abordados en este capítulo sigan evolucionando y cambiando, ya que se trata de un área donde la tecnología avanza rápidamente. Hemos abordado los fundamentos y mencionado las publicaciones fundamentales. En la siguiente sección se incluyen más referencias, pero se recomienda al lector mantenerse al tanto de las novedades en las publicaciones.

10.7 Notas bibliográficas

Las estadísticas sobre big data están dispersas y no existe una sola publicación que ofrezca estadísticas exhaustivas. Las estadísticas de YouTube provienen de [Brewer et al., 2016], mientras que las de Alibaba provienen de correspondencia personal. Se publican muchas más cifras en diversos blogs y publicaciones web.

Un buen análisis inicial de los problemas con la compatibilidad de los sistemas operativos con los SGBD se encuentra en Stonebraker [1981], que también explica por qué los primeros SGBD pasaron del almacenamiento basado en sistemas de archivos al almacenamiento en bloques. Entre los sistemas de almacenamiento más recientes que analizamos, Google File System se describe en [Ghemawat et al., 2003] y Ceph en [Weil et al., 2006].

Nuestro análisis de las plataformas de procesamiento de big data (en particular, MapReduce) se basa principalmente en Li et al. [2014]. Sakr et al. [2013] y Lee et al. [2012] también ofrecen una visión general del tema. La propuesta original de MapReduce se encuentra en [Dean y Ghemawat 2004, 2010]. Las críticas a MapReduce se discuten en [DeWitt et al.

2008, Dewitt y Stonebraker 2009, Pavlo y otros 2009, Stonebraker y otros 2010].

Las fuentes de los lenguajes MapReduce son las siguientes: HiveQL [Thusoo et al. 2009], Tenzing [Chattopadhyay et al. 2011], JAQL [Beyer et al. 2009], Pig Latin [Olston et al. 2008], Sawzall [Pike et al. 2005]), FlumeJava [Chambers et al. 2010] y SystemML [Ghoting et al. 2011]. El algoritmo 1-Bucket-Theta de implementación de unión de MapReduce se debe a Okcan y Riedewald [2011], la unión de difusión se debe a Blanas et al. [2010] y la unión de repartición es propuesta por Blanas et al. [2010].

Spark se propone en [Zaharia et al., 2010, Zaharia , 2016]. Como parte del ecosistema Spark, Spark SQL se analiza en [Armbrust et al., 2015], Spark Streaming en [Zaharia et al., 2013] y GraphX en [Gonzalez et al., 2014].

Sobre sistemas de flujo de datos, la rica literatura se encuentra en varios libros. Golab y Özsu [2010] se centran principalmente en los sistemas anteriores y en cuestiones de modelado de datos y consultas. El libro también aborda los almacenes de flujos. Aggarwal [2007] abarca una amplia gama de temas (incluida la minería de flujos, que omitimos) centrados en trabajos anteriores. Se puede encontrar más información sobre la minería de flujos en [Bifet et al., 2018].

Muthukrishnan [2005] se centra en los fundamentos teóricos de estos sistemas.

La generalización de los sistemas de flujo de datos al procesamiento de eventos es otra dirección que se ha seguido; aunque no hemos abordado este tema en este capítulo, Etzion y Niblett [2010] constituyen un buen punto de partida para investigar esta dirección. Además de los sistemas que analizamos, existen implementaciones de DSS en la nube, como lo demuestra StreamCloud [Gulisano et al. , 2010, 2012].

La definición de un flujo de datos como una secuencia de solo anexión de elementos con marca de tiempo que llegan en algún orden [Guha y McGregor 2006]. Otras definiciones de un flujo de datos se dan en [Wu et al. 2006, Tucker et al. 2003]. El concepto de tuplas de revisión se introduce en flujos de datos por Ryvkina et al. [2006]. La semántica de consulta de streaming se discute por Arasu et al. [2006] dentro del contexto del lenguaje CQL y más generalmente por Law et al. [2004]. Estos lenguajes se clasifican como declarativos (QL [Arasu et al. 2006, Arasu y Widom 2004], GSQL [Cranor et al. 2003] y StreaQuel [Chandrasekaran et al. 2003]) o procedimentales (Aurora [Abadi et al. 2003]). Las ejecuciones de operadores en sistemas de flujos son importantes debido a sus requisitos de no bloqueo. Las uniones no bloqueantes son temas de [Haas y Hellerstein 1999a, Urhan y Franklin 2000, Viglas et al. 2003, Wilschut y Apers 1991] y la agregación de [Hellerstein et al. 1997, Wang et al. 2003]. Las uniones de más de dos flujos (uniones multiflujo) se discuten en [Golab y Özsu 2003, Viglas et al. 2003] y las uniones de flujos con datos estáticos son el tema de [Balazinska et al. 2007]. El tema de las puntuaciones como medio de desbloqueo es presentado por

Tucker et al. [2003]. Las puntuaciones también pueden utilizarse para reducir la cantidad de estado que los operadores deben soportar [Ding y Rundensteiner 2004, Ding et al. 2004, Fernández-Moctezuma et al. 2009, Li et al. 2006, 2005]. Los latidos, que son puntuaciones que rigen las marcas de tiempo de futuras tuplas, se analizan en [Johnson et al. 2005, Srivastava y Widom 2004a].

El procesamiento de consultas sobre flujos de datos es el tema de [Abadi et al. 2003, Adamic y Huberman 2000, Arasu et al. 2006, Madden y Franklin 2002, Madden et al. 2002a]. El procesamiento de consultas en ventana se analiza en [Golab y Özsü 2003, Hammad et al. 2003a, 2005, Kang et al. 2003, Wang et al. 2004, Arasu et al.

Los enfoques de gestión de carga cuando la tasa de flujo excede la capacidad de procesamiento se presentan en [Tatbul et al. 2003, Srivastava y Widom 2004b, Ayad y Naughton 2004, Liu et al. 2006, Reiss y Hellerstein 2005, Babcock et al. 2002, Cammert et al. 2006, Wu et al. 2005].

Se han propuesto y desarrollado varios sistemas de procesamiento de flujos como prototipos y sistemas de producción. Algunos de ellos se clasifican como Sistemas de Gestión de Flujos de Datos (DSMS): STREAM [Arasu et al., 2006], Gigascope [Cranor et al., 2003], TelegraphCQ [Chandrasekaran et al., 2003], COUGAR [Bonnet et al.

2001], Tribeca [Sullivan y Heybey 1998], Aurora [Abadi et al. 2003], Bore-alis [Abadi et al. 2005]. Clasificamos otros como sistemas de procesamiento de flujo de datos (DSPS): Apache Storm [Toshniwal et al. 2014], Heron [Kulkarni et al. 2015], Spark Streaming [Zaharia et al. 2013], Flink [Carbone et al. 2015], MillWheel [Akidau et al. 2013] y TimeStream [Qian et al. 2013]. Como se mencionó, todos los DSMS, excepto Bore-alis, eran sistemas de una sola máquina, mientras que todos los DSPS son distribuidos/paralelos.

La partición de datos de streaming en sistemas paralelos/distribuidos se analiza en [Xing et al. 2006] y [Johnson et al. 2008]. La división de claves es propuesta por Azar et al. [1999]. Agrupación de Claves Parciales (PKG) de Nasir et al. [2015] (la "potencia de dos opciones" en la que se basa la PKG se analiza en [Mitzenmacher 2001]). La PKG se ha ampliado para utilizar más de dos opciones para la cabeza de la distribución [Nasir et al. 2016]. El particionamiento híbrido para gestionar datos sesgados se describe en [Gedik 2014, Pacaci y Özsü 2018]. El reparticionamiento entre operaciones en el plan de consulta se analiza en [Zhu et al. 2004, Elseidy et al. 2014, Heinze et al. 2015, Fernandez et al. 2013, Heinze et al. 2014]. En este contexto, Shah et al. [2003] proponen el flujo de trabajo seminal .

La semántica de recuperación de sistemas de flujo paralelo/distribuido es el tema de [Hwang et al. 2005].

Existen numerosos libros que se centran en los aspectos específicos de las plataformas de análisis de grafos, y estos suelen abordar cómo realizar análisis utilizando alguna de las plataformas que analizamos. Para un libro más general sobre procesamiento de grafos, [Desh-pande y Gupta 2018] es una buena fuente. El análisis de grafos se aborda en una extensa encuesta [Yan et al. 2017]. La encuesta de Larriba-Pey et al. [2014] también es una excelente referencia. Lumsdaine et al. analizan los verdaderos desafíos del procesamiento de grafos.

[2007]. McCune et al. [2015] proporcionan un buen estudio de los sistemas centrados en vértices.

Las características de los grafos, en particular la asimetría en la distribución de grados, desempeñan un papel importante en el procesamiento de grafos. Esto se analiza en [Newman et al., 2002]. Un primer paso importante en el procesamiento de grafos paralelo/distribuido es la partición de grafos, que implica

Consumo una gran parte del tiempo de procesamiento [Verma et al., 2017] y es computacionalmente costoso [Andreev y Racke, 2006]. Las técnicas de partición de grafos pueden ser de dos clases: disjuntas de vértices (corte de arista) y disjuntas de arista (corte de vértice). El algoritmo principal en la primera clase es METIS [Karypis y Kumar, 1995] cuyo costo de cómputo es analizado por McCune et al. [2015]. El hash es otra posibilidad cuando los vértices se dividen en diferentes particiones. Estas técnicas distribuyen los vértices de manera equilibrada, pero no manejan bien los grafos de ley de potencia. Se han desarrollado extensiones a METIS para este caso [Abou-Rjeili y Karypis, 2006]. Alternativamente, la propagación de etiquetas Ugander y Backstrom [2013] es un enfoque alternativo para abordar este problema. También se ha propuesto combinar METIS con la propagación de etiquetas incorporando esta última en la fase de engrosamiento de METIS [Wang et al. 2014]. Otra alternativa es partir de una partición desequilibrada y alcanzar el equilibrio gradualmente [Ugander y Backstrom, 2013]. Para particiones con aristas disjuntas, es posible aplicar hash. También es posible combinar enfoques con vértices disjuntos y con aristas disjuntas, como en PowerLyra [Chen et al., 2015].

MapReduce se ha propuesto como un posible enfoque para procesar grafos [Cohen 2009, Kiveris et al. 2014, Rastogi et al. 2013, Zhu et al. 2017], así como modificaciones que permitirían una mejor escalabilidad [Qin et al. 2014]. El sistema HaLoop [Bu et al. 2010, 2012] es un enfoque de MapReduce especialmente diseñado para el análisis de grafos. GraphX [Gonzalez et al. 2014] es un sistema basado en Spark que sigue el enfoque de MapReduce.

En sistemas nativos de analítica gráfica, la clasificación que analizamos en la sección 10.4.3 se basa en [Han 2015, Corbett et al. 2013]. El modelo de computación paralela síncrona en masa (BSP) se debe a Valiant [1990]. Los sistemas BSP centrados en vértices incluyen Pregel [Malewicz et al. 2010] y su contraparte de código abierto Apache Giraph [Apache], GPS [Salihoglu y Widom 2013], Mizan [Khayyat et al. 2013], LFGraph [Hoque y Gupta 2013], Pregelix [Bu et al. 2014] y Trinity [Shao et al. 2013].

Las optimizaciones a nivel de sistema para abordar la asimetría se analizan en [Lugowski et al., 2012; Salihoglu y Widom , 2013; Gonzalez et al., 2012]. Algunas optimizaciones algorítmicas se presentan en [Salihoglu y Widom, 2014]. Los sistemas asíncronos centrados en vértices incluyen GRACE [Wang et al., 2013] y GiraphUC [Han y Daudjee , 2015]. El principal ejemplo de sistemas de recopilación, aplicación y dispersión centrados en vértices es GraphLab [Low et al., 2012, 2010]. Blogel [Yan et al., 2014] y Giraph++ [Tian et al., 2013] siguen el enfoque BSP centrado en particiones. X-Stream [Roy et al., 2013] es el único sistema BSP centrado en aristas desarrollado hasta la fecha.

El lago de datos es un tema nuevo y, por lo tanto, es prematuro encontrar muchos libros técnicos al respecto. Pasupuleti y Purra [2015] ofrecen una buena introducción a las arquitecturas de lagos de datos, centrándose en la gobernanza, la seguridad y la calidad de los datos. También se puede encontrar información útil en informes técnicos, por ejemplo, [Hortonworks 2014] de empresas que ofrecen componentes y servicios para lagos de datos. Algunos de los desafíos que enfrentan los lagos de datos residen en la integración de big data. Dong y Srivastava [2015] ofrecen un excelente análisis de las técnicas recientes para la integración de big data. El tema más amplio de la integración de big data, que incluye la web, es el tema de [Dong y Srivastava 2015]. En este contexto, [Coletta et al. 2012]

Sugerimos combinar técnicas de aprendizaje automático, emparejamiento y agrupamiento para abordar problemas de integración en lagos de datos.

Ceremonias

Problema 10.1 Compare y contraste los diferentes enfoques para el diseño de sistemas de almacenamiento en términos de escalabilidad, facilidad de uso (ingesta de datos, etc.), arquitectura (compartida, sin compartir, etc.), esquema de consistencia, tolerancia a fallas, gestión de metadatos. Intente generar una tabla comparativa además de una breve discusión.

Problema 10.2 (*) Considere la pila de software de gestión de big data (Fig. 10.1) y compárela con la pila de software de un SGBD relacional tradicional, por ejemplo, basándose en la Fig. 1.9. En particular, analice las principales diferencias en cuanto a la gestión del almacenamiento.

Problema 10.3 (*) En la capa de almacenamiento distribuido de la pila de software de gestión de big data, los datos se suelen almacenar en archivos u objetos. Analice cuándo usar almacenamiento de objetos en función de las características de los datos (p. ej., objetos grandes o pequeños, gran cantidad de objetos, registros similares) y los requisitos de la aplicación (p. ej., facilidad de transferencia entre máquinas, escalabilidad y tolerancia a fallos).

Problema 10.4 (*) En un sistema de archivos distribuido como GFS o HDFS, los archivos se dividen en particiones de tamaño fijo, llamadas fragmentos. Explique las diferencias entre el concepto de fragmento y la partición horizontal, tal como se define en el capítulo 2.

Problema 10.5 Considere las diversas implementaciones de MapReduce para la unión equitativa descritas en la Sección 10.2.1.3. Compare la unión de difusión y la unión de repartición en términos de generalidad y costo de redistribución.

Problema 10.6 (*) En la sección 10.2.1.1 se describe cómo se utiliza el módulo de combinación para reducir el costo de mezcla.

(a) Proporcione el pseudocódigo de dicha función combinadora para el ejemplo de consulta SQL dado en el Ejemplo 10.1. (b)

Describa cómo reduciría el costo de mezcla.

Problema 10.7. Considere la implementación de MapReduce del operador de unión theta y su flujo de datos, como se muestra en la figura 10.10. Analice las implicaciones de rendimiento de dicho flujo de datos para la unión equitativa (donde θ es =).

Problema 10.8. Considere la implementación de Spark del algoritmo de agrupamiento k-medias presentado en el Ejemplo 10.3. Describa qué pasos del algoritmo resultan en la redistribución de datos entre múltiples trabajadores.

Problema 10.9 Analizamos el cálculo de PageRank en el Ejemplo 10.4. Una versión, llamada PageRank personalizado, calcula el valor de una página en torno a un conjunto de páginas seleccionado por el usuario asignando mayor importancia a los bordes en la vecindad de ciertas

Páginas que el usuario ha identificado. En esta pregunta, supongamos que el conjunto de páginas que el usuario ha identificado es una sola página, denominada página fuente. El cálculo se realiza con respecto a esta página fuente. Las diferencias con el PageRank habitual son las siguientes:

- Recordemos que en PageRank, cuando el recorrido aleatorio llega a una página, con una probabilidad de d , el recorrido salta a una página aleatoria del grafo. En PageRank personalizado, el salto no es a una página aleatoria, sino siempre a la página de origen; es decir, con una probabilidad de d , el recorrido regresa a la página de origen.
- Al inicializar el cálculo, en lugar de asignar valores de PageRank iguales a todos los vértices del grafo, a la página de origen se le asigna un rango de 1 y al resto de las páginas, un rango de 0.

Calcule (a mano) el PageRank personalizado del gráfico web que se muestra en la Fig. 10.21.

Problema 10.10 (**) Implemente PageRank personalizado como se define en el Problema 10.9 en MapReduce (use Hadoop).

Problema 10.11 (**) Implemente PageRank personalizado como se define en el Problema 10.9 en Spark.

Problema 10.12 (**) Considere un DSPS como el descrito en la Sección 10.3. Describa un algoritmo para implementar la semántica de entrega al menos una vez.

Problema 10.13 (**) Considere el DSS como se describe en la Sección 10.3.

(a) Diseñe una versión paralela intraoperador del operador de flujo de filtro. (b) Diseñe una versión paralela intraoperador del operador de agregación. Sugerencia: A diferencia del operador de filtro anterior, el operador de agregación sí tiene estado. ¿Qué debe tener en cuenta que no era necesario para el operador de filtro (sin estado)?
¿Cómo se dividen los datos entre las instancias del operador? ¿Qué se debe hacer en la salida del operador anterior para garantizar que cada tupla de streaming se dirija a la instancia correcta?

Problema 10.14 (**) Diseñe un operador de unión de ventana deslizante para dos flujos. ¿Es determinista? ¿Por qué no? ¿Podría proponer un diseño alternativo del operador que garantice el determinismo independientemente de la velocidad/entrelazado relativo de los flujos de entrada?

Problema 10.15 Considere el modelo de programación centrada en vértices para el procesamiento de grafos como se describe en la Sección 10.4.3. Compare los modelos de cálculo BSP y GAS en términos de

(a) generalidad y expresividad de los algoritmos gráficos y (b)
optimizaciones del rendimiento

Problema 10.16 (**) Proporcione un algoritmo para PageRank personalizado como se define en el Problema 10.9 utilizando el modelo BSP centrado en vértices.

Problema 10.17 (*) El ejemplo 10.8 describe un algoritmo iterativo basado en la propagación de etiquetas para encontrar componentes conexos del grafo de entrada en el modelo de programación centrada en vértices. Considere una aplicación de streaming donde cada tupla entrante en el flujo representa una arista no dirigida del grafo de entrada. Diseñe un algoritmo incremental que encuentre componentes conexos del grafo formado por las aristas del flujo.

Problema 10.18 (*) Considere las heurísticas voraces de colocación de aristas con corte de vértice definidas en la Sección 10.4.10. Se sabe que estas heurísticas voraces de partición presentan desequilibrio de carga si el flujo de aristas se presenta en un orden adverso.

(a) Cree un ordenamiento de los vértices en la Fig. 10.23 tal que la heurística descrita resulte en una partición altamente desequilibrada, es decir, todo el conjunto de aristas se asigne a una sola partición. (b)

Proponga una estrategia para mitigar el desequilibrio de carga en caso de tal ordenamiento de flujo adversario.

Problema 10.19 (*) Un lago de datos se asemeja a un almacén de datos (véase el capítulo 7), pero para datos no estructurados y sin esquema, por ejemplo, almacenados en HDFS. Considere el sistema de big data Spark, que proporciona acceso SQL a datos HDFS (a través de SparkSQL) y a muchas otras fuentes de datos. ¿Es Spark suficiente para construir un lago de datos? ¿Qué funcionalidad faltaría?

Problema 10.20 (**) Los sistemas de gestión de bases de datos (SGBD) paralelos recientes utilizados en los almacenes de datos modernos han añadido compatibilidad con tablas externas (véase, por ejemplo, Polybase en el capítulo 11), que establecen la correspondencia con los archivos HDFS y pueden manipularse junto con tablas relacionales nativas mediante consultas SQL. Por otro lado, los lagos de datos proporcionan acceso a fuentes de datos externas, como SQL, NoSQL, etc., mediante contenedores, como los conectores Spark. Compare los dos enfoques (lago de datos y almacén de datos moderno) desde el punto de vista de la integración de datos. ¿Qué similitudes hay? ¿Qué es diferente?

Capítulo 11

NoSQL, NewSQL y Polystores



Para gestionar datos en la nube, siempre se puede confiar en un SGBD relacional. Todos los SGBD relacionales tienen una versión distribuida y la mayoría operan en la nube.

Sin embargo, estos sistemas han sido criticados por su enfoque de "talla única".

Si bien han podido integrar compatibilidad con todo tipo de datos (p. ej., objetos multimedia, documentos) y nuevas funciones, esto ha resultado en una pérdida de rendimiento, simplicidad y flexibilidad para aplicaciones con requisitos de rendimiento específicos y estrictos. Por lo tanto, se ha argumentado que se necesitan motores DBMS más especializados. Por ejemplo, se ha demostrado que los DBMS orientados a columnas, que almacenan datos de columnas juntos en lugar de filas como los DBMS relacionales tradicionales, tienen un rendimiento mucho mejor en cargas de trabajo OLAP. De igual manera, los sistemas de gestión de flujos de datos (véase la sección 10.3) están diseñados específicamente para gestionar eficientemente los flujos de datos.

Por lo tanto, se han propuesto numerosas soluciones de gestión de datos, especializadas para distintos tipos de datos y tareas, capaces de ofrecer un rendimiento considerablemente superior al de los SGBD relacionales tradicionales. Entre las nuevas tecnologías de gestión de datos se incluyen los sistemas de archivos distribuidos y los marcos de procesamiento de datos paralelos para big data (véase el capítulo 10).

Una importante nueva tecnología de gestión de datos es NoSQL, que significa "No solo SQL" en contraste con el enfoque universal de los SGBD relacionales. Los sistemas NoSQL son almacenes de datos especializados que abordan los requisitos de la gestión de datos web y en la nube. El término "almacén de datos" se utiliza a menudo por su carácter general, e incluye no solo los SGBD, sino también sistemas de archivos o directorios más sencillos. Como alternativa a los SGBD relacionales, los sistemas NoSQL admiten diferentes modelos de datos y lenguajes además del SQL estándar. También priorizan la escalabilidad, la tolerancia a fallos y la disponibilidad, a veces en detrimento de la consistencia. Existen diferentes tipos de sistemas NoSQL, como los de clave-valor, de documento, de columna ancha y de grafos, así como los híbridos (multimodelo o NewSQL).

Estas nuevas tecnologías de gestión de datos han dado lugar a una amplia oferta de servicios que pueden utilizarse para crear aplicaciones en la nube con uso intensivo de datos que pueden escalar y exhibir

Alto rendimiento. Sin embargo, esto también ha conllevado una amplia diversificación de las interfaces de almacenamiento de datos y la pérdida de un paradigma de programación común. Por lo tanto, dificulta enormemente la creación de aplicaciones que utilicen múltiples almacenes de datos, como sistemas de archivos distribuidos, sistemas de gestión de bases de datos relacionales y sistemas de gestión de bases de datos NoSQL. Esto ha motivado el diseño de polialmacenes, también llamados sistemas multialmacenamiento, que proporcionan acceso integrado a varios almacenes de datos en la nube mediante uno o más lenguajes de consulta.

Este capítulo se organiza de la siguiente manera: la Sección 11.1 analiza las motivaciones de los sistemas NoSQL, en particular el teorema CAP, que ayuda a comprender el equilibrio entre diferentes propiedades. A continuación, se presentan los diferentes tipos de sistemas NoSQL: clave-valor en la Sección 11.2, documento en la Sección 11.3, columna ancha en la Sección 11.4 y grafo en la Sección 11.5. La Sección 11.6 presenta los sistemas híbridos, es decir, los sistemas NoSQL multimodelo y los SGBD NewSQL. La Sección 11.7 analiza los polialmacenes.

11.1 Motivaciones para NoSQL

Existen varias razones (complementarias) que han motivado la necesidad de los sistemas NoSQL. La primera, obvia, es la limitación de los SGBD relacionales, que ya comentamos, de que "son universales".

Una segunda razón es la limitada escalabilidad y disponibilidad de la arquitectura de bases de datos inicial utilizada en la nube. Esta arquitectura es tradicional de tres niveles, con clientes web que acceden a un centro de datos que cuenta con un balanceador de carga, servidores web/de aplicaciones y servidores de bases de datos. El centro de datos suele utilizar un clúster sin recursos compartidos, la solución más rentable para la nube. Para cada aplicación, existe un servidor de base de datos, generalmente un SGBD relacional, que proporciona tolerancia a fallos y disponibilidad de datos mediante replicación. A medida que aumenta el número de clientes web, resulta relativamente fácil añadir servidores web/de aplicaciones, generalmente mediante máquinas virtuales, para absorber la carga entrante y escalar. Sin embargo, el servidor de base de datos se convierte en un cuello de botella, y añadir nuevos servidores requeriría replicar toda la base de datos, lo que llevaría mucho tiempo. En un clúster sin recursos compartidos, una solución podría ser utilizar un SGBD relacional paralelo para proporcionar escalabilidad. Sin embargo, esta solución sólo sería apropiada para cargas de trabajo OLAP (lectura intensiva) (ver Sección 8.2) y no sería rentable ya que los DBMS relacionales paralelos son productos de alta gama.

Una tercera razón que se ha utilizado para justificar la necesidad de sistemas NoSQL es que el soporte de una consistencia de base de datos sólida, como lo hacen los SGBD relacionales, es decir, mediante transacciones ACID, perjudica la escalabilidad. Por lo tanto, algunos sistemas NoSQL han relajado la consistencia de base de datos sólida en favor de la escalabilidad. Un argumento para respaldar este enfoque ha sido el famoso teorema CAP de la teoría de sistemas distribuidos. Sin embargo, el argumento es simplemente erróneo, ya que el teorema CAP no tiene nada que ver con la escalabilidad de la base de datos: está relacionado con la consistencia de replicación en presencia de particiones de red. Además, es perfectamente posible proporcionar tanto una consistencia de base de datos sólida como escalabilidad, como lo hacen algunos sistemas NewSQL (véase la sección 11.6.2).

El teorema CAP establece que un almacén de datos distribuido con replicación solo puede proporcionar dos de las siguientes tres propiedades: (C) consistencia, (A) disponibilidad y (P) tolerancia a particiones. Cabe destacar que no existe (S) escalabilidad. Estas propiedades se definen de la siguiente manera:

- Consistencia: todos los nodos ven los mismos valores de datos simultáneamente; es decir, cada solicitud de lectura devuelve el último valor escrito. Esta propiedad se corresponde con la linealización (consistencia en operaciones individuales) y no con la serialización (consistencia en grupos de operaciones).
- Disponibilidad: cualquier réplica debe responder a cualquier solicitud recibida.
- Tolerancia a particiones: el sistema continúa funcionando a pesar de una partición de la red debido a un fallo.

Un error común en el teorema CAP es que se debe descartar una de estas propiedades. Sin embargo, solo en el caso de una partición de red se debe elegir entre consistencia y disponibilidad.

NoSQL (No Solo SQL) es un término confuso que deja mucho margen de interpretación y definición. Por ejemplo, puede aplicarse a los primeros DBMS jerárquicos y de red, o a los DBMS de objetos o XML. Sin embargo, el término surgió a finales de la década de 1990 para referirse a los nuevos almacenes de datos diseñados para satisfacer las necesidades de la gestión de datos web y en la nube. Como alternativa a las bases de datos relacionales, admiten modelos de datos y lenguajes distintos del SQL estándar. Estos sistemas suelen priorizar la escalabilidad, la tolerancia a fallos y la disponibilidad, a veces en detrimento de la consistencia.

En este capítulo, presentamos las cuatro categorías principales de sistemas NoSQL según el modelo de datos subyacente, es decir, clave-valor, columna ancha, documento y gráfico. También consideraremos los almacenes de datos híbridos: multimodelo, para combinar múltiples modelos de datos en un solo sistema, y NewSQL, para combinar la escalabilidad de NoSQL con la sólida consistencia de los SGBD relacionales. Para cada categoría, se ilustra con un sistema representativo.

11.2 Almacenes de clave-valor

En el modelo de datos clave-valor, todos los datos se representan como pares clave-valor, donde la clave identifica de forma única el valor. Los almacenes de clave-valor no requieren esquema, lo que proporciona gran flexibilidad y escalabilidad. Suelen ofrecer una interfaz sencilla, como put (clave, valor), value=get (clave), delete (clave).

Una forma extendida de almacenamiento clave-valor permite almacenar registros como listas de pares atributo-valor. El primer atributo se denomina clave principal o clave primaria (p. ej., un número de la seguridad social) e identifica de forma única el registro entre un conjunto de registros (p. ej., personas). Las claves suelen estar ordenadas, lo que permite realizar consultas por rango, así como su procesamiento ordenado.

Un almacén de clave-valor popular es Amazon DynamoDB, que presentamos a continuación.

11.2.1 DynamoDB

DynamoDB es utilizado por algunos de los servicios principales de Amazon que requieren alta disponibilidad y acceso a datos basado en claves. Algunos ejemplos de estos servicios son los que proporcionan carritos de compra, listas de vendedores, preferencias de clientes y catálogos de productos. Para lograr escalabilidad y disponibilidad, Dynamo sacrifica la consistencia en algunos escenarios de fallo y utiliza una síntesis de técnicas P2P conocidas (véase el capítulo 9) en un clúster sin recursos compartidos.

DynamoDB almacena datos como tablas de base de datos, que son colecciones de elementos individuales. Cada elemento es una lista de pares atributo-valor. Un valor de atributo puede ser de tipo escalar, conjunto o JSON. Los elementos son análogos a las filas de una tabla relacional, y los atributos a las columnas. Sin embargo, dado que los atributos se autodescriben, no se necesita un esquema relacional. Además, los elementos pueden ser heterogéneos, es decir, con diferentes atributos.

El diseño original de DynamoDB proporciona la abstracción de tabla hash distribuida (DHT) P2P (véase la sección 9.1.2). La clave principal (el primer atributo) se procesa mediante hash en las diferentes particiones, lo que permite operaciones eficientes de lectura y escritura basadas en claves en un elemento, así como el equilibrio de carga. Recientemente, DynamoDB se ha ampliado para admitir claves primarias compuestas, que se componen de dos atributos. El primer atributo es la clave hash y no es necesariamente único. El segundo atributo es la clave de rango y permite operaciones de rango dentro de la partición hash correspondiente a la clave hash. Para acceder a las tablas de la base de datos, DynamoDB proporciona una API de Java con las siguientes operaciones:

- PutItem, UpdateItem, DeleteItem: agrega, actualiza o elimina un elemento en una tabla según su clave principal (ya sea una clave principal hash o una clave principal compuesta).
- GetItem: devuelve un elemento según su clave principal en una tabla.
- BatchGetItem: devuelve todos los elementos que tienen la misma clave principal, pero en varios Tablas.
- Escaneo: devuelve todos los elementos de una tabla. • Consulta de rango: devuelve todos los elementos según una clave hash y un rango basado en la clave de rango. • Consulta indexada: devuelve todos los elementos según un atributo indexado.

Ejemplo 11.1. Considere la tabla Foro_Hilo de la Fig. 11.1. Esta tabla está compuesta por elementos homogéneos con cuatro atributos: Foro, Asunto, Fecha de la última publicación y Etiquetas. Tiene una clave compuesta, compuesta por una clave hash (Foro) y una clave de rango (Asunto).

Un ejemplo de acceso a clave principal es

Obtener elemento (Foro = "EC2", Asunto = "xyz")

que devuelve el último elemento. Un ejemplo de consulta de rango es

Consulta(Foro="S3," Asunto >"ac")

que devuelve el segundo y tercer elemento.

DynamoDB crea un índice hash desordenado sobre la clave hash, es decir, un DHT, y un índice de rango ordenado sobre la clave de rango (ordenada). Además, DynamoDB proporciona

Tabla: Hilo del foro

Foro	Sujeto	Fecha de última publicación	Etiquetas
"T3"	"abecedario"	"2017..."	"a" "b"
"T3"	"acd"	"2017..."	"do"
"T3"	"CBD"	"2017..."	"d" "e"
"RDS"	"xyz"	"2017..."	"F"
"EC2"	"abecedario"	"2017..."	"a" "e"
"EC2"	"xyz"	"2017..."	"F"

Picadillo Rango
llave llave

Fig. 11.1 Ejemplo de tabla de DynamoDB

Dos tipos de índices secundarios para permitir un acceso rápido a elementos basados en claves no clave. atributos: índices secundarios locales, para recuperar elementos dentro de una partición hash, es decir, elementos que tienen el mismo valor en su clave hash y en los índices secundarios globales, recuperar elementos de toda la tabla DynamoDB.

Los datos se partitionan y replican en varios nodos del clúster en varias bases de datos. centros, que proporcionan equilibrio de carga y alta disponibilidad. Particionado de datos. Se basa en un hash consistente, un esquema de hash popular que se ha utilizado en DHT. con geometría de anillo, p. ej., Cuerda (véase la Sección 9.1.2). El DHT se representa como un espacio de identificadores circular unidimensional, es decir, un "anillo", donde cada nodo del sistema es se le asigna un valor aleatorio dentro de este espacio que representa su posición en el anillo. Cada elemento se asigna a un nodo mediante el hash de la clave del elemento para obtener su posición en el nodo. anillo y luego encontrar el primer nodo en el sentido de las agujas del reloj con una posición más alta que el elemento posición. Por lo tanto, cada nodo se vuelve responsable del intervalo en el anillo entre Su nodo predecesor y él mismo. La principal ventaja del hash consistente es que La adición (uniones) y la eliminación (bajas/fallos) de nodos solo afectan a los nodos. vecino inmediato, sin impacto en otros nodos.

DynamoDB también explota el hash consistente para proporcionar alta disponibilidad, mediante Replicando cada elemento en n nodos, donde n es un parámetro configurado por el sistema. Cada elemento se le asigna un nodo coordinador, como se describió anteriormente, y se replica en el n – 1 Nodos sucesores en sentido horario. Por lo tanto, cada nodo es responsable del intervalo de la anillo entre su enésimo predecesor y él mismo.

Ejemplo 11.2 La figura 11.2 muestra un anillo con 6 nodos, cada uno nombrado por su posición. (valor hash). Por ejemplo, el nodo B es responsable del intervalo de valores hash (A,B). y el nodo A para el intervalo (F,A]. La operación put(c,v) produce un valor hash para La clave c entre A y B, por lo que el nodo B se hace responsable del elemento. Además, suponiendo que el parámetro de replicación n = 3, el artículo se replicaría en los nodos C y

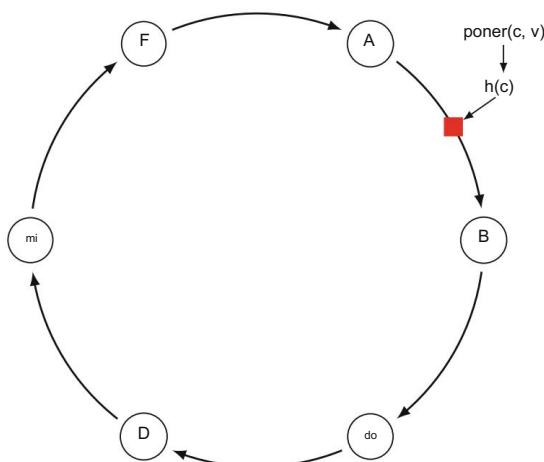


Fig. 11.2 Hashing consistente de DynamoDB. El nodo B es responsable del intervalo de valores hash (A,B). Por tanto, el elemento (c,v) se asigna al nodo B

D. De esta forma, el nodo D almacenará los elementos cuyas claves se encuentren en los intervalos (A, B], (B, C] y (C, D].

DynamoDB prioriza la alta consistencia de los datos a cambio de escalabilidad y disponibilidad, pero con diferentes métodos para controlar la consistencia. Proporciona consistencia final de las réplicas (véase la sección 6.1.1), lo cual se logra mediante un protocolo de propagación de actualizaciones asíncronas y un protocolo de detección de fallos distribuido basado en Gossip.

La consistencia de escritura en un entorno multiusuario concurrente se puede controlar mediante escrituras condicionales. De forma predeterminada, las operaciones de escritura (PutItem, UpdateItem, DeleteItem) sobrescribirán un elemento existente que tenga la clave principal dada. Una escritura condicional especifica una condición sobre los atributos del elemento para que se ejecute correctamente. Por ejemplo, una condición para que una escritura PutItem tenga éxito es que no exista ya un elemento con la misma clave principal. Por lo tanto, las escrituras condicionales son útiles en caso de actualizaciones concurrentes.

DynamoDB admite lecturas eventualmente consistentes y fuertemente consistentes. De forma predeterminada, las lecturas son eventualmente consistentes, es decir, pueden no devolver los datos más recientes que se replican asíncronamente. Las lecturas fuertemente consistentes devuelven los datos más actualizados, lo que podría no ser posible en caso de fallos de red, ya que no se han propagado todas las actualizaciones de las réplicas.

11.2.2 Otros almacenes de clave-valor

Otros almacenes de clave-valor populares son Cassandra, Memcached, Riak, Redis, Amazon SimpleDB y Oracle NoSQL Database. Muchos sistemas ofrecen extensiones adicionales.

para que podamos ver una transición suave hacia almacenes de columnas anchas y almacenes de documentos, que analizaremos a continuación.

11.3 Almacenes de documentos

Los almacenes de documentos son almacenes clave-valor avanzados, donde las claves se asignan a valores de tipo de documento, como JSON, YAML o XML. Los documentos suelen agruparse en colecciones, que desempeñan una función similar a la de las tablas relacionales. Sin embargo, los documentos son diferentes de las tuplas relativas. Los documentos se autodefinen, almacenando datos y metadatos (p. ej., marcado en XML, nombres de campo en objetos JSON) en conjunto y pueden ser diferentes entre sí dentro de una colección. Además, las estructuras de los documentos son jerárquicas y utilizan construcciones anidadas, como objetos anidados y matrices en JSON.

De esta forma, modelar una base de datos usando documentos requiere menos colecciones que con tablas relativas (planas) y también evita costosas operaciones de unión.

Además de la sencilla interfaz clave-valor para recuperar documentos, los almacenes de documentos ofrecen una API o lenguaje de consulta que recupera documentos según su contenido. Los almacenes de documentos facilitan la gestión de cambios y valores opcionales, así como su mapeo a objetos de programa. Esto los hace atractivos para las aplicaciones web modernas, sujetas a cambios continuos y donde la velocidad de implementación es crucial.

Un almacén de documentos NoSQL popular es MongoDB, que presentamos a continuación.

11.3.1 MongoDB

MongoDB es un sistema de código abierto escrito en C++. Ofrece un modelo de datos basado en JSON para documentos, flexibilidad de esquema, alta disponibilidad, tolerancia a fallos y escalabilidad en clústeres sin recursos compartidos.

MongoDB almacena datos como documentos en BSON (JSON binario), una serialización de JSON codificada en binario que incluye tipos adicionales como binario, entero, largo y punto flotante. Los documentos BSON contienen uno o más campos, cada uno con un nombre y un valor de un tipo de dato específico, incluyendo matrices, datos binarios y subdocumentos. Cada documento es un objeto BSON, es decir, con múltiples campos, identificado de forma única por su primer campo de tipo ObjectId, cuyo valor es generado automáticamente por MongoDB.

Los documentos con una estructura similar se organizan como colecciones, como las tablas relativas, donde los campos de los documentos son similares a las columnas. Sin embargo, los documentos de una misma colección pueden tener estructuras diferentes, ya que no existe un esquema predefinido.

MongoDB proporciona un lenguaje de consulta completo para actualizar y recuperar datos BSON mediante funciones expresadas en JSON. Representar las consultas como JSON permite unificar el almacenamiento y la manipulación de los datos. Este lenguaje de consulta se puede utilizar con API en diversos lenguajes de programación, como Java, PHP, JavaScript y Scala.

Dado que las consultas se implementan como métodos o funciones dentro de la API de un lenguaje de programación específico, la integración dentro de los programas de aplicación es natural y sencilla para los desarrolladores.

MongoDB admite diversos tipos de consultas para insertar, actualizar, eliminar y recuperar documentos. Una consulta puede devolver documentos, subconjuntos de campos específicos dentro de documentos o agregaciones complejas de valores de varios documentos. Las consultas también pueden incluir funciones JavaScript definidas por el usuario. La forma general de una consulta es

```
db.collection.function (expresión JSON)
```

Donde db es una variable global que corresponde a la conexión a la base de datos, y función es la operación de base de datos aplicada a la colección. La expresión JSON puede ser arbitraria y usarse como criterio para seleccionar datos. Los diferentes tipos de consultas son

- Operaciones de inserción, eliminación y actualización en documentos. Para las operaciones de eliminación y actualización, la expresión JSON especifica los criterios para seleccionar los documentos.
- Las consultas de coincidencia exacta devuelven resultados basados en la igualdad de un valor para un campo en los documentos, normalmente la clave principal.

Las consultas de rango devuelven resultados basados en los valores de un campo en un rango determinado. Las consultas geoespaciales devuelven resultados basados en la proximidad, la intersección y la inclusión de objetos geográficos, como puntos, líneas, círculos o polígonos en formato GeoJSON.

Las consultas de búsqueda de texto devuelven resultados ordenados por relevancia según los argumentos de texto mediante operadores booleanos. Las consultas de agregación devuelven valores agregados de una colección con operadores como recuento, mínimo, máximo y promedio. Además, los documentos de dos colecciones se pueden combinar mediante una operación de unión externa izquierda.

Ejemplo 11.3. Considere la colección "Publicaciones" de la Fig. 11.3. Cada elemento de publicación de la colección se identifica de forma única mediante su clave de tipo ObjectId (generada por MongoDB) y su valor es un objeto JSON, con matrices anidadas como etiquetas y comentarios. Ejemplos de operaciones de actualización son:

```
db.posts.insert(autor:"alex," título:"No hay almuerzo gratis")
db.posts.update(autor:"alex," $set:age:30)
db.posts.update(autor:"alex," $push:tags:"música")
```

Clave única generada por MongoDB	Valor = objeto JSON con matrices anidadas
_id: ObjectId("abc")	autor: "alex", título: "No hay almuerzo gratis", texto: "Esto es...", etiquetas: ["negocios", "divagaciones"], comentarios: [{"quién: "jane", qué: "Estoy de acuerdo."}, {"quién: "joe", qué: "No..." }]
_id: ObjectId("abd")	Una publicación de X
_id: ObjectId("acd")	Una publicación de Y

Fig. 11.3 Ejemplo de colección de publicaciones de MongoDB

donde \$set (establece un valor específico para un campo) y \$push (agrega un valor específico a una matriz) son instrucciones MongoDB dentro de JSON.

Las siguientes consultas:

```
db.posts.find(autor:"alex")
db.posts.find(comentarios.quién:"jane")
```

Son consultas de coincidencia exacta. La primera devuelve todas las publicaciones de Alex, mientras que la segunda devuelve todas las publicaciones en las que Jane comentó.

Para facilitar el acceso a los datos, MongoDB admite diversos tipos de índices secundarios que pueden declararse en cualquier campo del documento, incluidos los campos dentro de matrices. Los diferentes tipos de índices son:

- Índices únicos, donde el valor del campo indexado se exige que sea único. • Índices multicluve (compuestos) en varios campos. • Índices de matriz para campos de matriz, con una entrada de índice independiente para cada valor de matriz. • Índices TTL que expiran automáticamente después de un cierto tiempo de vida (TTL). • Índices geoespaciales para optimizar las consultas según la proximidad, la intersección y la inclusión de objetos geográficos, como puntos, líneas, círculos o polígonos. • Índices parciales que se crean para un subconjunto de documentos que cumplen una condición. especificado por el usuario.
- Índices dispersos que indexan solo los documentos que contienen el campo especificado. • Los índices de búsqueda de texto utilizan reglas lingüísticas específicas del idioma para optimizar la búsqueda de texto. consultas.

Para escalar horizontalmente en clústeres sin recursos compartidos, MongoDB admite diferentes esquemas de particionamiento de datos (o fragmentación): basados en hash, basados en rangos y con reconocimiento de ubicación (donde el usuario especifica los rangos de claves y los nodos asociados). Se proporciona alta disponibilidad mediante una variante de la replicación de copia principal, denominada conjuntos de réplicas, con propagación de actualizaciones asíncrona. Si una réplica maestra falla, una de las réplicas se convierte en la nueva maestra y continúa aceptando operaciones de actualización. Un clúster MongoDB consta de: fragmentos (particiones de datos), donde cada fragmento puede ser una unidad de replicación (un conjunto de réplicas); mongo, que actúan como procesadores de consultas entre las aplicaciones cliente y el clúster; y servidores de configuración que almacenan metadatos y opciones de configuración para el clúster. Las aplicaciones pueden leer opcionalmente desde réplicas secundarias, donde los datos son consistentes.

MongoDB ha introducido recientemente compatibilidad con transacciones ACID en múltiples documentos, además de transacciones de un solo documento. Esto se logra mediante el aislamiento de instantáneas. Uno o más campos de un documento pueden escribirse en una sola transacción, incluyendo actualizaciones de múltiples subdocumentos y elementos de una matriz.

Las transacciones multidocumento se pueden utilizar en múltiples colecciones, bases de datos, documentos y fragmentos.

MongoDB también ofrece una opción llamada preocupación por la escritura , que permite a los usuarios especificar un nivel de garantía para informar del éxito de una operación de escritura, es decir, con un equilibrio deseado entre el nivel de persistencia y el rendimiento, en las réplicas de la base de datos. Existen cuatro niveles de garantía para que los clientes ajusten el control de una

._id:ObjectId("abc") autor:	alex", título: "No hay almuerzo gratis", texto: "Esto es...", etiquetas: ["negocios", "divagaciones"], comentarios: [quién: "jane", qué: "Estoy de acuerdo.", quién: "joe", qué: "No..."]
._id: ObjectId("abd") Una publicación de	
._X id: ObjectId("acd") Una publicación de Y	

Clave única generada
por MongoDB

Valor = objeto JSON con matrices anidadas

Figura 11.4 Arquitectura de MongoDB

Operación de escritura, ordenada de la más débil a la más fuerte. Estas son: no reconocida (sin garantía), reconocida (la escritura en el disco se realizó), registrada (la escritura se registró en el registro) y reconocida por réplica (la escritura se propagó a las réplicas).

La arquitectura de MongoDB se integra en la pila de software de gestión de big data (véase la Fig. 10.1). Admite motores de almacenamiento conectables, como HDFS o en memoria, para gestionar las demandas específicas de las aplicaciones, interactúa con frameworks de big data como MapReduce y Spark, y es compatible con herramientas de terceros para análisis, IoT, aplicaciones móviles, etc. (véase la Fig. 11.4). Utiliza ampliamente la memoria principal para acelerar las operaciones de la base de datos y la compresión nativa mediante su motor de almacenamiento (WiredTiger).

11.3.2 Otros almacenes de documentos

Otros almacenes de documentos populares son AsterixDB, Couchbase, CouchDB y RavenDB, que admiten el modelo de datos JSON en una arquitectura de clúster escalable sin recursos compartidos. Sin embargo, AsterixDB y Couchbase admiten un dialecto de SQL++, una elegante extensión de SQL con algunas funciones sencillas para consultar datos JSON. El dialecto SQL++ de Couchbase se denomina N1QL (lenguaje de consulta de forma no normal, que se pronuncia "nickel"). Couchbase admite transacciones restringidas (escrituras atómicas de documentos) y permite intercambiar algunas propiedades por rendimiento, por ejemplo, reduciendo la durabilidad al reconocer las operaciones de escritura realizadas en memoria y luego escribiendo asincrónicamente en el disco. Además del servidor Couchbase, que se utiliza para consultas y transacciones, la plataforma Couchbase cuenta con un servicio de análisis que permite el análisis de datos en línea, sin afectar el rendimiento de las transacciones y sin requerir un modelo de datos diferente ni (como resultado) ETL. Este es un tipo de Procesamiento Híbrido de Transacciones y Análisis (HTAP) para NoSQL (véase la introducción de HTAP en la Sección 11.6.2). La implementación de este servicio de análisis se basa en el motor de almacenamiento y el procesador de consultas paralelas de AsterixDB. AsterixDB es un almacén de documentos JSON de alto rendimiento que combina técnicas de bases de datos paralelas y bases de datos documentales.

11.4 Almacenes de columnas anchas

Los almacenes de columnas anchas combinan algunas de las propiedades de las bases de datos relacionales (p. ej., la representación de datos como tablas) con la flexibilidad de los almacenes clave-valor (p. ej., datos sin esquema dentro de columnas). Cada fila de una tabla de columnas anchas se identifica de forma única mediante una clave y tiene varias columnas con nombre. Sin embargo, a diferencia de una tabla relacional, donde las columnas solo pueden contener valores atómicos, una columna puede ser ancha y contener múltiples pares clave-valor.

Los almacenes de columnas amplias amplían la interfaz del almacén de valores clave con más construcciones declarativas que permiten exploraciones, coincidencias exactas y consultas de rango sobre familias de columnas. Suelen proporcionar una API para usar estas construcciones en un lenguaje de programación. Algunos sistemas también ofrecen un lenguaje de consulta similar a SQL, por ejemplo, Cassandra Query Language (CQL).

En el origen de las tiendas de columnas anchas se encuentra Google Bigtable, que presentamos a continuación.

11.4.1 Tabla grande

Bigtable es un almacén de columnas amplio para clústeres sin recursos compartidos. Bigtable utiliza el Sistema de Archivos de Google (GFS) para almacenar datos estructurados en archivos distribuidos, lo que proporciona tolerancia a fallos y disponibilidad (véase la sección 10.1 sobre sistemas de archivos distribuidos basados en bloques). También utiliza un método de partición dinámica de datos para mejorar la escalabilidad. Al igual que GFS, lo utilizan aplicaciones populares de Google, como Google Earth, Google Analytics y Google+.

Bigtable admite un modelo de datos simple similar al modelo relacional, con atributos multivalor con marca de tiempo. Describiremos brevemente este modelo, ya que constituye la base de la implementación de Bigtable, que combina aspectos de sistemas de gestión de bases de datos (SGBD) con almacenamiento en filas y columnas. Para mantener la coherencia con los conceptos utilizados hasta ahora, presentamos el modelo de datos de Bigtable como un modelo relacional ligeramente ampliado.¹

Una instancia de Bigtable es una colección de pares (clave-valor) donde la clave identifica una fila y el valor es el conjunto de columnas, organizadas como familias de columnas. Bigtable ordena sus datos por claves, lo que facilita la agrupación de filas del mismo rango en el mismo nodo de clúster.

Cada fila de una Bigtable se identifica de forma única mediante una clave de fila, que es una cadena arbitraria (de hasta 64 KB en el sistema original). Por lo tanto, una clave de fila es como una clave de atributo única en una relación. Una Bigtable siempre se ordena por claves de fila. Una fila puede tener varias familias de columnas, que constituyen la unidad de control de acceso y almacenamiento. Una familia de columnas es un conjunto de columnas del mismo tipo. Para crear una Bigtable, solo es necesario especificar el nombre de la tabla y el nombre de la familia de columnas. Sin embargo, dentro de una familia de columnas, se pueden añadir dinámicamente columnas arbitrarias (del mismo tipo).

¹En la propuesta original, una Bigtable se define como un mapa multidimensional, indexado por una clave de fila, una clave de columna y una marca de tiempo, siendo cada celda del mapa un valor único (una cadena).

Nombre de la clave de fila	Correo	Página web
100 "Prefijo": "Dr." "Último": "Dobb"	electrónico "email: gmail.com": "dobb@gmail.com"	<!DOCTYPE html PÚBLICO...>
101 "Primero": "Alicia" Apellido: Martin, correo electrónico: free.fr	"correo electrónico: gmail.com": "amartin@gmail.com" "correo electrónico: free.fr": "amartin@free.fr"	<!DOCTYPE html PÚBLICO...>

Fig. 11.5 Una tabla grande con 3 familias de columnas y 2 filas

Para acceder a los datos en una Bigtable, es necesario identificar columnas dentro de la columna Familias que utilizan claves de columna. Una clave de columna es un nombre completo con la forma nombre-de-familia-de-columnas:nombre-de-columnas. El nombre de la familia de columnas es como una relación. Nombre del atributo. El nombre de la columna es como un valor de atributo de relación, pero se usa como nombre, como parte de la clave de columna para representar un solo elemento. Esto permite el equivalente de atributos multivalor dentro de una relación. Además, los datos identificados por una columna La clave dentro de una fila puede tener varias versiones, cada una identificada por una marca de tiempo (un 64 bit entero).

Ejemplo 11.4 La figura 11.5 muestra un ejemplo de una Bigtable con 3 familias de columnas y 2 filas, como representación de estilo relacional. La columna Nombre y Correo electrónico Las familias tienen columnas heterogéneas. Para acceder al valor de una columna, se utiliza la clave de fila, y se debe especificar la clave de columna, p. ej., clave de fila = "111" y clave de columna = "Correo electrónico:gmail.com" que da como resultado "am@gmail.com".

Bigtable proporciona una API básica para definir y manipular tablas, dentro de una Lenguaje de programación como C++. También proporciona funciones para cambiar tablas, y metadatos de familias de columnas, como los derechos de control de acceso. La API ofrece varias operadores para escribir y actualizar valores, y para iterar sobre subconjuntos de datos, producidos por un operador de escaneo. Hay varias formas de restringir las filas, columnas y Marcas de tiempo producidas por un escaneo, como en un operador de selección relacional. Sin embargo, existen No hay operadores complejos como join o union, que deben programarse utilizando El operador del escáner.

La atomicidad transaccional solo se admite para actualizaciones de una sola fila. Por lo tanto, para más En el caso de actualizaciones complejas de varias filas, es responsabilidad del programador escribir la información adecuada. Código para controlar la atomicidad usando una interfaz para agrupar escrituras en las claves de fila los clientes.

Para almacenar una tabla en GFS, Bigtable utiliza la partición de rango en la clave de fila. Cada La tabla se divide en particiones llamadas tabletas, cada una correspondiente a un rango de filas. La partición es dinámica y comienza con una tabla (todo el rango de tablas) que es Posteriormente se divide en varias tabletas a medida que la tabla crece. Para localizar al (usuario) En GFS, Bigtable utiliza una tabla de metadatos, que a su vez está particionada en metadatos. tabletas, con una única tabla raíz almacenada en un servidor maestro, similar al maestro de GFS. Además de explotar GFS para escalabilidad y disponibilidad, Bigtable utiliza varios técnicas para optimizar el acceso a los datos y minimizar el número de accesos al disco, como como compresión de familias de columnas, agrupación de familias de columnas con alta localidad de acceso y almacenamiento en caché agresivo de información de metadatos por parte de los clientes.

Bigtable se basa en un servicio de bloqueo distribuido persistente y de alta disponibilidad llamado Chubby.

Un servicio Chubby consta de cinco réplicas activas, una de las cuales se elige como maestra y atiende activamente las solicitudes. Bigtable utiliza Chubby para diversas tareas: garantizar que haya como máximo una maestra activa en todo momento; almacenar la ubicación de arranque de los datos de Bigtable; descubrir servidores de tabletas y finalizar la eliminación de servidores de tabletas; y almacenar esquemas de Bigtable. Si Chubby deja de estar disponible durante un período prolongado, Bigtable deja de estarlo.

11.4.2 Otros almacenes de columnas anchas

Existen implementaciones populares de código abierto de Bigtable, como Hadoop Hbase, una implementación popular de Java que se ejecuta sobre HDFS y Cassandra que combina técnicas de Bigtable y DynamoDB.

11.5 DBMS gráficos

En el capítulo 10, presentamos el análisis de grafos , donde el grafo completo puede procesarse varias veces hasta alcanzar un punto fijo. Por el contrario, los SGBD de grafos admiten consultas no iterativas que podrían acceder solo a una parte del grafo, lo que permite la eficacia de los índices. Las bases de datos de grafos representan y almacenan datos directamente como grafos, lo que facilita la expresión y el procesamiento rápido de consultas de tipo grafo, por ejemplo, calcular la ruta más corta entre dos elementos del grafo. Esto es mucho más eficiente que con una base de datos relacional, donde los datos de grafos deben almacenarse como tablas separadas y las consultas de tipo grafo requieren operaciones de unión repetidas y costosas. Los SGBD de grafos suelen proporcionar un potente lenguaje de consulta de grafos. Se han popularizado en aplicaciones web con uso intensivo de datos, como redes sociales y sistemas de recomendación.

Los sistemas de gestión de bases de datos (SGBD) de grafos pueden proporcionar un esquema flexible al especificar los tipos de vértices y aristas con sus propiedades. Esto facilita la definición de índices para un acceso rápido a los vértices, basándose en el valor de una propiedad (por ejemplo, el nombre de una ciudad), además de los índices estructurales. Las consultas de grafos pueden expresarse mediante operadores de grafos mediante una API específica o un lenguaje de consulta declarativo.

Como hemos visto anteriormente, para escalar a bases de datos muy grandes, los almacenes de clave-valor, documentos y columnas anchas dividen los datos en particiones en varios nodos del clúster.

La razón por la que esta partición de datos funciona bien es que trata elementos individuales. Sin embargo, la partición de datos de grafos es mucho más difícil, ya que el problema de particionar óptimamente un grafo es NP-completo (véase la sección 10.1). En particular, recuerde que podemos obtener elementos en diferentes particiones conectados por aristas.

Recorrer estas aristas entre particiones genera sobrecarga de comunicación, lo que perjudica el rendimiento de las operaciones de recorrido de grafos. Por lo tanto, se debe minimizar el número de aristas entre particiones. Sin embargo, esto también puede generar particiones desequilibradas.

A continuación, ilustramos DBMS gráficos con Neo4j, que es un sistema popular en muchas implementaciones.

11.5.1 Neo4j

Neo4j es un sistema comercial de código abierto presentado como un DBMS gráfico escalable y de alto rendimiento con almacenamiento y procesamiento nativo de grafos en clústeres sin recursos compartidos. Ofrece un modelo de datos gráfico completo con restricciones de integridad, un potente lenguaje de consulta llamado Cypher con índices, transacciones ACID y compatibilidad con alta disponibilidad y balanceo de carga.

El modelo de datos se basa en grafos dirigidos, con almacenamiento separado para aristas (llamadas relaciones), vértices (llamados nodos) o atributos (llamados propiedades). Cada nodo puede tener cualquier número de propiedades, en forma de pares (atributo, valor). Una relación debe tener un tipo, que le da semántica, y una dirección de un nodo a otro (o a sí misma). Una capacidad importante de Neo4j es que las relaciones se pueden recorrer en ambas direcciones con el mismo rendimiento. Esto simplifica el modelado de DBMS de grafos, ya que no hay necesidad de crear dos relaciones diferentes entre nodos, si una implica a la otra, p. ej., una relación mutua como amigo (cuyo reverso también es amigo) o una relación 1-1 como posee (cuyo reverso es propiedad de).

Actualizar un gráfico implica actualizar nodos, relaciones y propiedades, lo que Debe realizarse de forma consistente. Esto se logra mediante transacciones ACID.

Ejemplo 11.5 La figura 11.6 muestra un ejemplo de un gráfico simple para una red social. La relación de amistad entre Bob y Mary (que significa «Bob es amigo de Mary») es suficiente para representar la relación mutua (con suerte, Mary también es amiga de Bob). También podría haberse representado a la inversa (de Mary a Bob). Esto simplifica el modelo gráfico.

La siguiente transacción, utilizando la API de Java, crea los nodos Bob y Mary, sus propiedades y la relación de amistad de Bob a Mary.

```
Transacción tx = neo.beginTx(); Nodo n1 =
neo.createNode(); n1.setProperty("nombre",
"Bob"); n1.setProperty("edad", 35); Nodo n2 =
neo.createNode(); n2.setProperty("nombre",
"María"); n1.setProperty("edad", 29);
n1.setProperty("trabajo", "ingeniero");
n1.createRelationshipTo(n2, RelTypes.amigo);
tx.Commit();
```

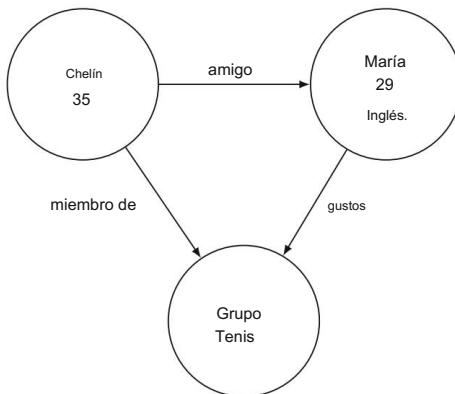


Fig. 11.6 Ejemplo de gráfico de Neo4j

Neo4j no impone ningún esquema en el grafo, lo que proporciona gran flexibilidad al permitir la creación de datos sin necesidad de comprender completamente de antemano cómo se utilizarán. Sin embargo, los esquemas en otros modelos de datos (relacionales, de objetos, XML, etc.) han demostrado ser útiles para la consistencia de la base de datos y el procesamiento eficiente de consultas.

Por lo tanto, Neo4j introduce un esquema opcional basado en el concepto de etiquetas y un lenguaje de definición de datos para manipularlo. Una etiqueta es similar a una etiqueta, útil para agrupar nodos similares. A un nodo se le puede asignar cualquier número de etiquetas, por ejemplo, persona, estudiante y usuario. Esto permite a Neo4j consultar solo un subconjunto del grafo, por ejemplo, los estudiantes de una ciudad determinada. Las etiquetas se utilizan para definir restricciones de integridad e índices.

Se pueden definir restricciones de integridad en nodos y relaciones, por ejemplo, propiedad de nodo única, existencia de propiedad de nodo o existencia de propiedad de relación.

Se pueden crear índices basados en etiquetas y combinaciones de propiedades. Proporcionan una búsqueda eficiente de nodos, una operación importante para iniciar recorridos de grafos en nodos específicos según predicados que involucran etiquetas y propiedades.

La biblioteca espacial Neo4j también proporciona índices poligonales de n dimensiones para optimizar las consultas geoespaciales.

Para consultar y manipular datos de grafos, Neo4j proporciona una API de Java y un lenguaje de consulta llamado Cypher. La API de Java permite al programador acceder a las operaciones de grafos de nodos, relaciones, propiedades y etiquetas, con transacciones ACID. Esta API proporciona una integración estrecha con el lenguaje de programación.

Cypher es un potente lenguaje de consulta para SGBD de grafos con variante SQL. Permite manipular datos de grafos. Por ejemplo, la transacción del Ejemplo 11.5 podría escribirse simplemente con la siguiente sentencia CREATE :

```
CREAR (:Persona {nombre:"Bob", edad:35}) <- [:AMIGO]
-(:Persona {nombre:"María", edad:29, trabajo:"ingeniero"})
```

Cypher es fácil de usar a través de la coincidencia de patrones de gráficos: el usuario especifica un patrón de gráfico como cuando dibuja un diagrama y consulta la base de datos para

Encuentra los datos que coinciden con el patrón. Cypher proporciona cláusulas como MATCH, MERGE, WHERE y RETURN, que pueden manipular variables de nodo (como las variables de tupla en SQL) y algunas otras. MATCH realiza la coincidencia de patrones de grafos.

Los nodos se expresan con paréntesis, y las relaciones, mediante pares de guiones con signos de mayor o menor que, indican la dirección de la relación. Los pares clave-valor de las propiedades de nodo y relación se especifican entre llaves. MERGE es útil para crear o unir grafos. WHERE especifica un predicado sobre los nodos y RETURN los nodos, las relaciones y las propiedades resultantes.

Ejemplo 11.6. La siguiente consulta Cypher devuelve todos los amigos directos e indirectos de Bob cuyo nombre empieza por "M". La expresión MATCH define un patrón recursivo para la relación de amigo a amigo con las variables de nodo bob y seguidor. La expresión WHERE busca el nodo cuyo nombre es "Bob", lo cual puede hacerse mediante un índice, y selecciona los nodos seguidores cuyo nombre empieza por "M".

Las expresiones RETURN devuelven todos los pares de nodos vinculados a las variables bob y follower.

```
COINCIDIR bob-[:AMIGO]-> ()-[:AMIGO] -> seguidor DONDE bob.nombre = "Bob" Y
seguidor.nombre =~ "M.*"
REGRESAR bob, seguidor.nombre
```

Neo4j ofrece un compilador de consultas de bajo costo que genera planes de consulta optimizados para consultas Cypher, tanto de solo lectura como de actualización. El compilador primero reescribe lógicamente el plan de consulta mediante la desanidación, la fusión y la simplificación de varias partes de la consulta. Posteriormente, basándose en información estadística sobre la selectividad de índices y etiquetas, selecciona los mejores métodos de acceso para los operadores y genera el plan de ejecución de la consulta, utilizando iteradores anidados que se ejecutan de forma segmentada y descendente.

Neo4j ofrece un amplio soporte para alta disponibilidad mediante replicación completa, tanto a nivel de clúster como entre centros de datos. Dentro de un clúster, se utiliza una variante de la replicación multamaestro (véase el capítulo 6), denominada agrupación causal, para escalar horizontalmente a configuraciones de gran tamaño. La agrupación causal admite la consistencia causal, un modelo de consistencia que garantiza que todas las aplicaciones cliente vean las operaciones relacionadas causalmente en el mismo orden. De esta forma, se garantiza que una aplicación cliente lea sus propias escrituras.

La arquitectura de agrupamiento causal se muestra en la Fig. 11.7, con tres tipos de nodos de clúster: servidor de aplicaciones, servidor central y servidor de lectura. Los servidores de aplicaciones ejecutan código de aplicación y emiten transacciones de escritura a los servidores centrales y consultas de lectura a los servidores de lectura. Los servidores centrales replican todas las transacciones de forma asíncrona mediante el protocolo Raft. Raft garantiza la durabilidad de las transacciones, ya que la mayoría de los servidores centrales confirman una escritura antes de que se confirme de forma segura. Los servidores centrales replican las transacciones a los servidores de lectura mediante el envío de registros de transacciones. El protocolo Raft también se utiliza para implementar diversas arquitecturas de replicación en centros de datos para facilitar la recuperación ante desastres.

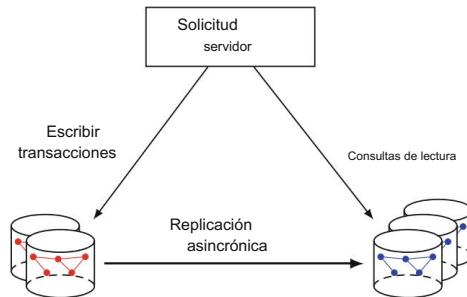


Fig. 11.7 Arquitectura de agrupamiento causal de Neo4j

Además de la alta disponibilidad, la agrupación causal permite escalar horizontalmente las consultas de grafos utilizando múltiples servidores de lectura. Para optimizar el uso de la memoria RAM, Neo4j emplea fragmentación basada en caché, que exige que todas las consultas del mismo usuario se envíen siempre al mismo servidor de lectura. Esto mejora la localización de referencia en cada servidor de lectura y permite el escalado a grafos muy grandes.

De lo anterior se desprende que el tamaño máximo de un grafo de base de datos está limitado al del disco duro de un servidor central. Esta restricción permite un rendimiento lineal en los recorridos de rutas, a la vez que exige optimizar al máximo el almacenamiento compacto de grafos. Neo4j utiliza la compresión dinámica de punteros para ampliar el espacio de direcciones disponible según sea necesario, a la vez que permite localizar los nodos y relaciones adyacentes de un nodo mediante un salto de puntero. Finalmente, la separación del almacenamiento de Neo4j para nodos, relaciones y propiedades permite una mayor optimización. Esto permite que los dos primeros almacenes conserven únicamente información básica y tengan registros de nodos y relaciones de tamaño fijo, lo que produce recorridos de rutas eficientes de $O(1)$. El almacén de propiedades permite registros de longitud dinámica.

11.5.2 Otras bases de datos de gráficos

Otros DBMS gráficos populares son Infinite Graph, Titan, GraphBase, Trinity y Sparksee.

11.6 Almacenes de datos híbridos

Los almacenes de datos híbridos combinan capacidades que suelen encontrarse en diferentes almacenes de datos y sistemas de gestión de bases de datos (SGBD). Distinguimos entre sistemas NoSQL multimodelo y SGBD NewSQL.

11.6.1 Almacenes NoSQL multimodelo

Los sistemas NoSQL multimodelo están diseñados para reducir la necesidad de gestionar múltiples sistemas al crear aplicaciones complejas. Ilustramos almacenes NoSQL multimodelo con OrientDB, un popular almacén de datos NoSQL que combina conceptos de modelos de datos de documentos y grafos NoSQL orientados a objetos. Otros sistemas multimodelo populares son ArangoDB y Microsoft Azure Cosmos DB.

OrientDB se originó como una implementación en Java de la capa de almacenamiento del DBMS Orientado a Objetos (inicialmente escrito en C++) para clústeres sin recursos compartidos. Proporciona un modelo de datos completo con esquemas, un potente lenguaje de consulta basado en SQL, transacciones ACID optimistas y compatibilidad con alta disponibilidad y balanceo de carga.

El modelo de datos es un modelo de datos gráfico, con conexiones directas entre registros. Existen cuatro tipos de registros: Documento, Bytes de Registro (datos binarios), Vértice y Arista. Cuando OrientDB genera un registro (la unidad de almacenamiento más pequeña), le asigna un identificador único, llamado ID de Registro.

El lenguaje de consulta es una extensión de SQL con recorridos de rutas de grafos. Admite diferentes tipos de índices: SB-Tree, que es el índice predeterminado; índice hash para consultas de coincidencia exacta eficientes; índice de texto completo de Lucene para búsquedas basadas en texto; e índice espacial de Lucene para consultas espaciales.

La gestión de esquemas se deriva de la orientación a objetos con herencia de clases. Una clase define un conjunto de registros similares y puede ser sin esquema, con esquema completo (como en bases de datos orientadas a objetos) o con esquema híbrido. El modo de esquema híbrido permite que las clases definan algunos atributos, pero algunos registros pueden tener atributos específicos. La herencia de clases se basa en la estructura; es decir, una subclase extiende una clase padre, heredando todos sus atributos.

Las clases son la base para agrupar y particionar registros en múltiples nodos. Cada clase puede tener una o más particiones, llamadas clústeres. Al insertar un nuevo registro en una clase, OrientDB selecciona el clúster donde almacenarlo mediante una de las siguientes estrategias preconfiguradas:

- predeterminado: selecciona el clúster utilizando un identificador de clúster predeterminado especificado en la clase;
- round-robin: organiza los clústeres para la clase en secuencia y asigna cada nuevo registro al siguiente clúster en orden;
- equilibrado: verifica la cantidad de registros en los clústeres para la clase y asigna el nuevo registro al clúster más pequeño;
- local: cuando se replica la base de datos, selecciona el clúster maestro en el nodo actual (que está procesando la inserción).

OrientDB admite la replicación multimaestro; es decir, todos los nodos del clúster sin recursos compartidos pueden escribir en la base de datos en paralelo. Las transacciones se procesan mediante control de concurrencia multiversión optimista, partiendo del supuesto de que existen pocos conflictos de actualización. Por lo tanto, las transacciones se procesan sin necesidad de consultarlas hasta el momento de confirmación. Cuando se confirma una transacción, se comprueba cada versión del registro para detectar actualizaciones conflictivas de otra transacción, lo que podría provocar la cancelación de algunas transacciones.

11.6.2 Nuevos DBMS de SQL

NewSQL es una clase reciente de SGBD que busca combinar la escalabilidad de los sistemas NoSQL con la alta consistencia y usabilidad de los SGBD relacionales. El objetivo principal es abordar los requisitos de los sistemas de información empresarial, que han sido respaldados por los SGBD relacionales tradicionales, pero que también necesitan escalabilidad. Los sistemas NoSQL proporcionan escalabilidad, así como disponibilidad, esquemas flexibles y API prácticas para la programación de aplicaciones complejas con uso intensivo de datos.

Como hemos visto en las secciones anteriores, esto se logra generalmente aprovechando la partición de datos en clústeres de servidores básicos sin recursos compartidos y reduciendo la consistencia de la base de datos. Por otro lado, los SGBD relacionales proporcionan una sólida consistencia de la base de datos con transacciones ACID y facilitan el uso de herramientas y aplicaciones con SQL estándar. También pueden escalar, utilizando su versión paralela, aunque generalmente a un coste elevado, incluso utilizando un clúster sin recursos compartidos.

Una clase importante de NewSQL es el procesamiento híbrido de transacciones y análisis (HTAP), cuyo objetivo es realizar OLAP y OLTP en los mismos datos.

HTAP permite realizar análisis en tiempo real sobre datos operacionales, evitando así la separación tradicional entre base de datos operacional y almacén de datos y la complejidad de tratar con ETL.

Los sistemas NewSQL son recientes y tienen arquitecturas diferentes. Sin embargo, podemos identificar las siguientes características comunes: modelo de datos relacionales y SQL estándar; transacciones ACID; escalabilidad mediante particionamiento de datos en clústeres sin recursos compartidos; y disponibilidad mediante replicación de datos.

En el resto de esta sección, ilustramos NewSQL con Google F1 y LeanXcale. Otros tipos de sistemas NewSQL son Apache Ignite, CockroachDB, Esgyn, GridGain, MemSQL, NuoDB, Splice Machine, VoltDB y SAP HANA.

11.6.2.1 F1

F1 es un sistema NewSQL de Google que combina la escalabilidad de Bigtable con la consistencia y usabilidad de los sistemas de gestión de bases de datos relacionales. Se diseñó para ser compatible con la aplicación AdWords, una aplicación con un alto volumen de actualizaciones. F1 proporciona un modelo de datos relacional con algunas extensiones, compatibilidad total con consultas SQL, índices, consultas ad hoc y transacciones optimistas. Está basado en Spanner, un sistema de almacenamiento de datos escalable para clústeres sin recursos compartidos (véase la sección 5.5.1).

El modelo de datos F1 es relacional, con una implementación jerárquica inspirada en Bigtable. Varias tablas relacionales, con dependencias de claves foráneas, pueden organizarse como una relación anidada, donde las filas de cada tabla secundaria se agrupan con las filas de su tabla principal según la clave de unión. Esto optimiza la actualización de varias filas de la misma clave foránea y acelera el procesamiento de las uniones. F1 también admite columnas de tabla con tipos de datos estructurados mediante búferes de protocolo, el mecanismo extensible e independiente del lenguaje de Google para serializar datos estructurados.

El uso de buffers de protocolo facilita la escritura de transformaciones entre filas de bases de datos y estructuras de datos en memoria.

La interfaz principal es SQL, que se utiliza tanto para transacciones OLTP como para consultas OLAP extensas. Extiende el SQL estándar con construcciones para acceder a los datos almacenados en las columnas de Protocol Buffer. F1 también permite combinar datos de Spanner con otras fuentes de datos, como archivos Bigtable y CSV. F1 admite una interfaz clave-valor NoSQL con acceso rápido a las filas mediante consultas de coincidencia exacta y de rango, y actualizaciones basadas en la clave principal. Los índices secundarios se almacenan en tablas de Spanner, cuya clave se concatena mediante la clave del índice y la clave principal de la tabla indexada.

Los índices F1 pueden ser locales o globales. Los índices locales son locales a una jerarquía de tablas e incluyen en sus claves de índice la clave principal de la fila raíz como prefijo. Sus entradas de índice se cubican con las filas que indexan, lo que optimiza las actualizaciones de índice. Por el contrario, los índices globales son globales a varias tablas y no incluyen la clave principal de la fila raíz como prefijo. Por lo tanto, no se pueden cubicar con las filas que indexan.

F1 admite la ejecución de consultas tanto centralizada como distribuida. La ejecución centralizada se utiliza para consultas OLTP cortas, donde una consulta completa se ejecuta en un nodo del servidor F1. La ejecución distribuida se utiliza para consultas OLAP, con un alto grado de paralelismo, mediante técnicas de reparticionamiento y streaming basadas en hash.

En la sección [5.5.1](#), presentamos el enfoque de Spanner para la gestión de transacciones escalables. Spanner también ofrece tolerancia a fallos, particionamiento de datos dentro de los centros de datos, replicación geográfica síncrona entre centros de datos y transacciones ACID.

En Spanner, a cada transacción se le asigna una marca de tiempo de confirmación que se utiliza para la ordenación global de las confirmaciones. F1 admite tres tipos de transacciones, además del sólido soporte de Spanner:

- Transacciones de instantáneas, para transacciones de solo lectura con semántica de aislamiento de instantáneas. tics, utilizando marcas de tiempo de instantáneas de Spanner.
- Transacciones pesimistas, utilizando transacciones basadas en bloqueo ACID proporcionadas por Llave.
- Transacciones optimistas, con una fase de lectura que no toma bloqueos y luego una fase de validación que detecta conflictos a nivel de fila, utilizando las marcas de tiempo de la última modificación de las filas, para decidir si confirmar o abortar.

11.6.2.2 LeanXcale

LeanXcale es un sistema NewSQL/HTAP con soporte completo para SQL y polystore en un clúster sin recursos compartidos. Cuenta con tres subsistemas principales: motor de almacenamiento, motor de consultas y motor transaccional; los tres distribuidos y altamente escalables (es decir, hasta cientos de nodos).

LeanXcale proporciona funcionalidad SQL completa sobre tablas relacionales con columnas JSON. Los clientes pueden acceder a LeanXcale con cualquier herramienta de análisis mediante un controlador JDBC. Una capacidad importante de LeanXcale es el acceso a polystores mediante el mecanismo de scripting del lenguaje de consulta CloudMdsQL (véase la sección [11.7.3.2](#)). Los almacenes de datos accesibles abarcan desde archivos de datos sin procesar distribuidos (p. ej., HDFS) hasta...

bases de datos SQL paralelas, a bases de datos NoSQL (por ejemplo, MongoDB, donde las consultas se pueden expresar como programas JavaScript).

El motor de almacenamiento es un almacén de clave-valor relacional propietario, KiVi, que permite una partición horizontal eficiente de tablas e índices, basándose en la clave principal o la clave de índice. Cada tabla se almacena como una tabla KiVi, donde la clave corresponde a la clave principal de la tabla LeanXcale y todas las columnas se almacenan tal como están en las columnas KiVi. Los índices también se almacenan como tablas KiVi, donde las claves de índice se asignan a las claves principales correspondientes. Este modelo permite una alta escalabilidad de la capa de almacenamiento mediante la partición de tablas e índices en los nodos de datos KiVi. KiVi proporciona las operaciones típicas de put y get de los almacenes de clave-valor, así como todas las operaciones de tabla única, como la selección basada en predicados, la agregación, la agrupación y la ordenación, es decir, cualquier operador algebraico excepto join. Las operaciones multitabla, es decir, joins, son realizadas por el motor de consultas y cualquier operador algebraico por encima de join en el plan de consultas. De esta forma, todos los operadores algebraicos debajo de una unión se envían al motor de almacenamiento KiVi.

El motor de consultas procesa cargas de trabajo OLAP sobre datos operativos, de modo que las consultas analíticas se responden sobre datos en tiempo real. La implementación paralela del motor de consultas sigue el enfoque de un solo programa y múltiples datos (SPMD), que combina paralelismo entre consultas e intraoperador. Con SPMD, varios trabajadores simétricos (hilos) en diferentes instancias de consulta ejecutan la misma consulta/operador, pero cada uno gestiona diferentes porciones de datos.

El motor de consultas optimiza las consultas mediante la optimización en dos pasos. A medida que se reciben las consultas, todos los trabajadores transmiten y procesan los planes de consulta. Para la ejecución en paralelo, se añade un paso de optimización que transforma un plan de consulta secuencial generado en uno paralelo. Esta transformación implica reemplazar los escaneos de tabla por escaneos de tabla paralelos y añadir operadores de reorganización para garantizar que, en operadores con estado (como agrupar por o unir), las filas relacionadas sean gestionadas por el mismo trabajador. Los escaneos de tabla paralelos dividen las filas de las tablas base entre todos los trabajadores; es decir, cada trabajador recupera un subconjunto disjunto de las filas durante el escaneo. Esto se logra dividiendo las filas y programando los subconjuntos obtenidos en las diferentes instancias del motor de consulta. Cada trabajador procesa las filas obtenidas de los subconjuntos programados en su instancia del motor de consulta, intercambiando filas con otros trabajadores según lo determinado por los operadores de mezcla añadidos al plan de consulta. Para procesar las uniones, el motor de consulta admite dos estrategias de intercambio de datos (mezcla y difusión) y varios métodos de unión (hash, bucle anidado, etc.), que se ejecutan localmente en cada trabajador después del intercambio de datos.

El motor de consultas está diseñado para integrarse con almacenes de datos arbitrarios, donde los datos residen en su formato natural y pueden recuperarse (en paralelo) mediante la ejecución de scripts específicos o consultas declarativas. Esto lo convierte en un potente polystore que puede procesar datos desde su formato original, aprovechando al máximo tanto el scripting expresivo como el paralelismo masivo. Además, se pueden aplicar uniones en cualquier conjunto de datos nativo, como HDFS o MongoDB, incluyendo tablas LeanXcale, aprovechando algoritmos eficientes de unión paralela. Para habilitar las consultas ad hoc de un conjunto de datos arbitrario, el motor de consultas procesa las consultas en el lenguaje de consulta CloudMdsQL, donde los scripts se encapsulan como subconsultas nativas (Sección 11.7.3.2).

En la sección [5.5.2](#), presentamos el enfoque de LeanXcale para la gestión de transacciones escalable. LeanXcale escala la gestión transaccional descomponiendo las propiedades ACID y escalando cada una de ellas de forma independiente, pero de forma componible. El motor transaccional proporciona una alta consistencia con aislamiento de instantáneas. Por lo tanto, las lecturas no se ven bloqueadas por las escrituras, utilizando el control de concurrencia multiversión. Admite la ordenación basada en marcas de tiempo y la detección de conflictos justo antes de la confirmación. El algoritmo distribuido para proporcionar consistencia transaccional puede confirmar transacciones completamente en paralelo sin ninguna coordinación mediante una separación inteligente de tareas. De esta manera, la visibilidad de los datos confirmados se separa del procesamiento de la confirmación. De esta manera, el procesamiento de la confirmación puede adoptar un enfoque completamente paralelo sin comprometer la consistencia, que se regula por la visibilidad de las actualizaciones confirmadas. Por lo tanto, las confirmaciones se realizan en paralelo, y siempre que haya un prefijo más largo de transacciones confirmadas sin interrupciones, la instantánea actual avanza a ese punto.

11.7 Polystores

Los polystores proporcionan acceso integrado a múltiples almacenes de datos en la nube, como NoSQL, sistemas de gestión de bases de datos relacionales o HDFS. Normalmente solo admiten consultas de solo lectura, ya que gestionar transacciones distribuidas entre almacenes de datos heterogéneos es un problema complejo. Podemos dividir los polialmacenes según su nivel de acoplamiento con los almacenes de datos subyacentes: débilmente acoplados, fuertemente acoplados e híbridos. En esta sección, presentamos para cada clase un conjunto de sistemas representativos, con su arquitectura y procesamiento de consultas. Finalizamos la sección con algunas observaciones.

11.7.1 Polialmacenes acoplados de forma flexible

Los polialmacenes débilmente acoplados se asemejan a los sistemas multibase de datos, ya que pueden gestionar almacenes de datos autónomos, accesibles a través de la interfaz común del polialmacenador, así como por separado a través de su API local. Siguen la arquitectura mediador-encapsulador con varios almacenes de datos (p. ej., NoSQL y SGBD relacionales), como se muestra en la figura [11.8](#). Cada almacén de datos es autónomo, es decir, se controla localmente y otras aplicaciones pueden acceder a él. La arquitectura mediador-encapsulador, utilizada en sistemas de integración de datos, permite escalar a un gran número de almacenes de datos.

Hay dos módulos principales: un procesador de consultas y un contenedor por almacén de datos. El procesador de consultas cuenta con un catálogo de almacenes de datos, y cada contenedor tiene un catálogo local de su almacén de datos. Una vez creados los catálogos y los contenedores, el procesador de consultas puede comenzar a procesar las consultas de entrada de los usuarios interactuando con los contenedores. El procesamiento típico de consultas es el siguiente:

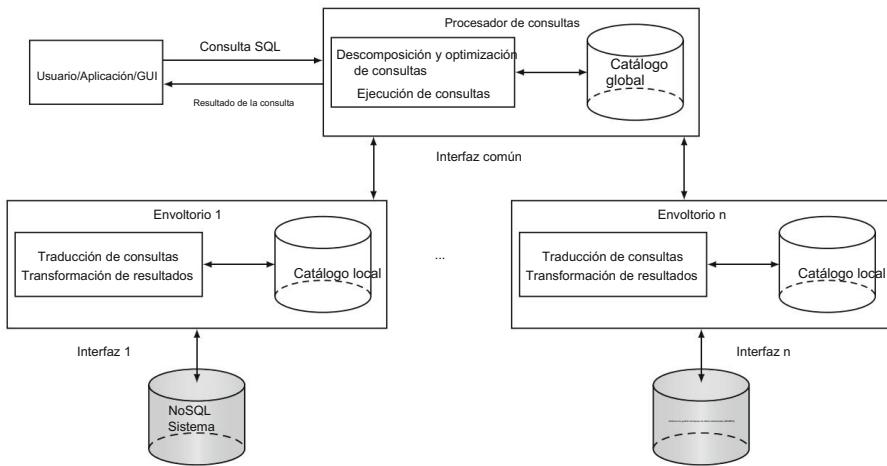


Fig. 11.8 Arquitectura de polalmacén débilmente acoplado

1. Analice la consulta de entrada y tradúzcala en subconsultas (una por almacén de datos), cada una expresado en un lenguaje común y una subconsulta de integración.
2. Envíe las subconsultas a los contenedores relevantes, que activan la ejecución en los almacenes de datos correspondientes y traducen los resultados al formato de lenguaje común.
3. Integrar los resultados de los wrappers (lo que puede implicar la ejecución de operadores como union y join) y devolver los resultados al usuario. A continuación, se describen tres polystores débilmente acoplados: BigIntegrator, Forward y QoX.

11.7.1.1 BigIntegrator

BigIntegrator admite consultas tipo SQL y combina datos de almacenes de datos de Bigtable en la nube con datos de sistemas de gestión de bases de datos relacionales (no necesariamente en la nube). Se accede a Bigtable mediante el lenguaje de consulta de Google (GQL), que tiene expresiones de consulta muy limitadas; por ejemplo, no permite uniones y solo predicados de selección básicos. Para aprovechar las capacidades limitadas de GQL, BigIntegrator proporciona un mecanismo de procesamiento de consultas basado en complementos, denominados absorbedor y finalizador, que permiten preprocesar y posprocesar operaciones que Bigtable no puede realizar. Por ejemplo, un predicado de selección "LIKE" en una Bigtable o una unión de dos Bigtables se procesará mediante operaciones en el procesador de consultas de BigIntegrator.

BigIntegrator utiliza el enfoque Local-Como-Vista (LAV) (véase la Sección 7.1.1) para definir el esquema global de Bigtable y las fuentes de datos relacional como tablas relacionales planas. Cada Bigtable o fuente de datos relacional puede contener varias colecciones, cada una representada como una tabla de origen con la forma "nombre-tabla_nombre-origen", donde nombre-tabla es el nombre de la tabla en el esquema global y nombre-origen es el nombre.

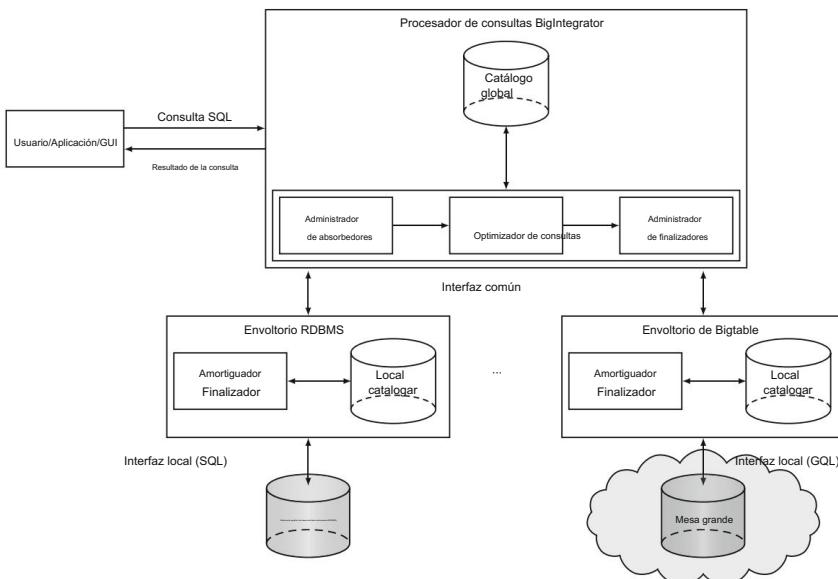


Fig. 11.9 Arquitectura de BigIntegrator

del origen de datos. Por ejemplo, "Empleados_A" representa una tabla Empleados en el origen A, es decir, una vista local de Empleados. Las tablas de origen se refieren como tablas en las consultas SQL.

La Figura 11.9 ilustra la arquitectura de BigIntegrator con dos fuentes de datos: una base de datos relacional y un almacén de datos Bigtable. Cada contenedor incluye un módulo importador y complementos de absorción y finalización. El importador crea las tablas de origen y las almacena en el catálogo local. Absorber extrae una subconsulta, denominada filtro de acceso, de una consulta de usuario que selecciona datos de una tabla de origen específica, según las capacidades de la fuente. Convierte cada filtro de acceso (generado por Absorber) en un operador denominado función de interfaz, específico para cada tipo de fuente. La función de interfaz se utiliza para enviar una consulta a la fuente de datos (es decir, una consulta GQL o SQL).

El procesamiento de consultas se realiza en tres pasos, utilizando un gestor de absorción, un optimizador de consultas y un gestor de finalizadores. El gestor de absorción toma la consulta de usuario (analizada) y, para cada tabla de origen referenciada en la consulta, llama al absorbente correspondiente de su contenedor. Para reemplazar la tabla de origen con un filtro de acceso, el absorbente recopila de la consulta las tablas de origen y los demás predicados posibles, según las capacidades de la fuente de datos. El optimizador de consultas reordena los filtros de acceso y otros predicados para generar una expresión algebraica que contiene llamadas tanto a filtros de acceso como a otros operadores relacionales. También realiza transformaciones tradicionales como la inserción de selección y la unión de enlace. El gestor de finalizadores toma la expresión algebraica y, para cada operador de filtro de acceso en la expresión a

Llama al finalizador correspondiente de su contenedor. El finalizador transforma los filtros de acceso en llamadas a funciones de interfaz.

Finalmente, la ejecución de la consulta la realiza el procesador de consultas que interpreta la expresión algebraica, llamando a las funciones de interfaz para acceder a las diferentes fuentes de datos y ejecutando las operaciones relacionales posteriores, utilizando técnicas en memoria.

11.7.1.2 Adelante

Forward es compatible con SQL++, un lenguaje similar a SQL, diseñado para unificar el modelo de datos y las capacidades de consulta de las bases de datos NoSQL y relacionales. SQL++ cuenta con un potente modelo de datos semiestructurado que amplía tanto los modelos de datos JSON como los relacionales. Forward también ofrece un completo framework de desarrollo web que aprovecha su compatibilidad con JSON para integrar componentes de visualización (p. ej., Google Maps).

El diseño de SQL++ se basa en la observación de que los conceptos son similares en ambos modelos de datos; por ejemplo, una matriz JSON es similar a una tabla SQL con orden, y una tupla SQL a un literal de objeto JSON. Por lo tanto, una colección SQL++ es una matriz o un contenedor, que puede contener elementos duplicados. Una matriz está ordenada (similar a una matriz JSON) y cada elemento es accesible por su posición ordinal, mientras que un contenedor no está ordenado (similar a una tabla SQL). Además, SQL++ extiende el modelo relacional con la composición arbitraria de valores complejos y la heterogeneidad de elementos. Al igual que en los modelos de datos anidados, un valor complejo puede ser una tupla o una colección. Se puede acceder a las colecciones anidadas anidando expresiones SELECT en la cláusula FROM de SQL o componiéndolas con el operador GROUP BY . También se pueden desanidar con el operador FLATTEN . A diferencia de una tabla SQL, que requiere que todas las tuplas tengan los mismos atributos, una colección SQL++ también puede contener elementos heterogéneos que comprenden una combinación de tuplas, escalares y colecciones anidadas.

Forward utiliza el enfoque Global-As-View (GAV) (véase la sección 7.1.1), donde cada fuente de datos (SQL o NoSQL) se presenta al usuario como una vista virtual de SQL++, definida sobre colecciones de SQL++. De esta forma, el usuario puede ejecutar consultas SQL++ que involucran múltiples vistas virtuales. La arquitectura de Forward es la de la figura 11.8, con un procesador de consultas y un contenedor por fuente de datos. El procesador de consultas realiza la descomposición de consultas SQL++, aprovechando al máximo las capacidades del almacén de datos subyacente. Sin embargo, dada una consulta SQL++ que no es compatible directamente con la fuente de datos subyacente, Forward la descompone en una o más consultas nativas compatibles y combina los resultados de la consulta nativa para compensar la brecha semántica o de capacidades entre SQL++ y la fuente de datos subyacente.

La optimización basada en costos de las consultas SQL++ es posible mediante la reutilización de técnicas de sistemas multibase de datos al trabajar con colecciones planas. Sin embargo, resultaría mucho más difícil considerando las capacidades de anidamiento y heterogeneidad de elementos de SQL++.

11.7.1.3 QoX

QoX es un tipo especial de polystore débilmente acoplado, donde las consultas son flujos de trabajo analíticos basados en datos (o flujos de datos) que integran datos de bases de datos relacionales y diversos motores de ejecución como MapReduce o herramientas ETL. Un flujo de datos típico puede combinar datos no estructurados (p. ej., tuits) con datos estructurados y utilizar operaciones genéricas de flujo de datos como filtrado, unión y agregación, así como funciones definidas por el usuario como análisis de sentimientos e identificación de productos. Un enfoque novedoso para el diseño ETL incorpora un conjunto de métricas de calidad, denominado QoX, en todas las etapas del proceso de diseño. El Optimizador de QoX gestiona las métricas de rendimiento de QoX con el objetivo de optimizar la ejecución de flujos de datos que integran tanto el flujo de integración ETL de back-end como las operaciones de consulta de front-end en un único flujo de análisis.

El Optimizador QoX utiliza xLM, un lenguaje propietario basado en XML para representar flujos de datos, generalmente creados con alguna herramienta ETL. xLM permite capturar la estructura del flujo, con nodos que muestran operaciones y almacenes de datos, y los bordes que los interconectan, así como propiedades operativas importantes como el tipo de operación, el esquema, las estadísticas y los parámetros. Mediante el uso de envoltorios adecuados para traducir xLM a un formato XML específico de la herramienta y viceversa, el Optimizador QoX puede conectarse a motores ETL externos e importar o exportar flujos de datos hacia y desde estos motores.

Dado un flujo de datos para múltiples almacenes de datos y motores de ejecución, el Optimizador QoX evalúa planes de ejecución alternativos, estima sus costos y genera un plan físico (código ejecutable). El espacio de búsqueda de planes de ejecución equivalentes se define mediante transformaciones del flujo de datos que modelan el envío de datos (desplazamiento de los datos al lugar donde se ejecutará la operación), el envío de funciones (desplazamiento de la operación al lugar donde se encuentran los datos) y la descomposición de la operación (en operaciones más pequeñas). El costo de cada operación se estima con base en estadísticas (p. ej., cardinalidades y selectividades). Finalmente, el Optimizador QoX genera código SQL para motores de bases de datos relacionales, código Pig y Hive para motores MapReduce, y crea scripts de shell de Unix como código de enlace necesario para orquestar los diferentes subflujos que se ejecutan en distintos motores. Este enfoque también podría extenderse al acceso a motores NoSQL, siempre que existan interfaces y wrappers similares a SQL.

11.7.2 Polialmacenes estrechamente acoplados

Los polystores estrechamente acoplados buscan la consulta eficiente de datos estructurados y no estructurados para el análisis de big data. También pueden tener un objetivo específico, como el autoajuste o la integración de datos HDFS y DBMS relacionales. Sin embargo, todos sacrifican autonomía por rendimiento, generalmente en un clúster sin recursos compartidos, de modo que solo se puede acceder a los almacenes de datos a través del polystore.

Al igual que los sistemas débilmente acoplados, proporcionan un lenguaje único para la consulta de datos estructurados y no estructurados. Sin embargo, el procesador de consultas utiliza directamente las interfaces locales del almacén de datos (véase la figura 11.10) o, en el caso de HDFS, puede interactuar con...

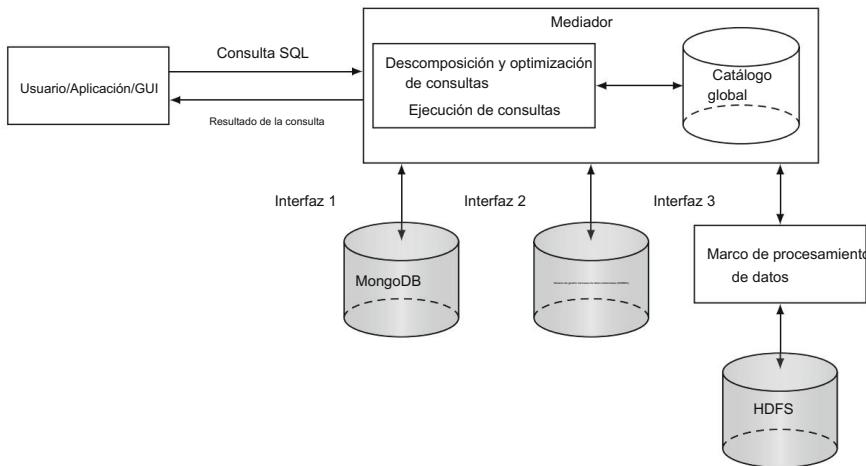


Fig. 11.10 Arquitectura de polialmacén estrechamente acoplado

Un marco de procesamiento de datos como MapReduce o Spark. Por lo tanto, durante la ejecución de la consulta, el procesador de consultas accede directamente a los almacenes de datos. Esto permite una transferencia eficiente de datos entre ellos. Sin embargo, el número de almacenes de datos que se pueden conectar suele ser muy limitado.

En el resto de esta sección, describimos tres polystores estrechamente acoplados representativos: Polybase, HadoopDB y Estocada. Otros tres sistemas interesantes son Redshift Spectrum, Odyssey y JEN. Amazon Redshift Spectrum es una función del producto de almacenamiento de datos Redshift de Amazon en la plataforma en la nube Amazon Web Services (AWS). Esta función permite ejecutar consultas SQL en grandes datos no estructurados que residen en Amazon Simple Storage Service (S3). Odyssey es un polystore compatible con diferentes motores analíticos, como sistemas OLAP paralelos o Hadoop. Permite almacenar y consultar datos en HDFS y DBMS relacionales mediante vistas materializadas oportunistas basadas en MISO, un método para ajustar el diseño físico de un polystore (Hive/HDFS y DBMS relacionales), es decir, decidir en qué almacén de datos deben residir los datos para mejorar el rendimiento del procesamiento de consultas de big data. Los resultados intermedios de la ejecución de consultas se tratan como vistas materializadas oportunistas, que pueden colocarse en los almacenes subyacentes para optimizar la evaluación de consultas posteriores. JEN es un componente sobre HDFS que proporciona un acoplamiento estrecho con un DBMS relacional paralelo. Permite unir datos de dos almacenes de datos, HDFS y un DBMS relacional, mediante algoritmos de unión paralelos, en particular un eficiente algoritmo de unión en zigzag, y técnicas para minimizar el movimiento de datos. A medida que aumenta el tamaño de los datos, ejecutar la unión en HDFS parece ser más eficiente.

11.7.2.1 Polybase

Polybase es una función de Microsoft SQL Server Parallel Data Warehouse (PDW), que permite a los usuarios consultar datos no estructurados (HDFS) almacenados en un clúster Hadoop mediante SQL e integrarlos con datos relacionales en PDW. Los datos HDFS se pueden referenciar en Polybase como tablas externas, que se corresponden con el archivo HDFS en el clúster Hadoop y, por lo tanto, se pueden manipular junto con las tablas nativas de PDW mediante consultas SQL. Polybase aprovecha las capacidades de PDW, un DBMS paralelo sin recursos compartidos. Mediante el optimizador de consultas de PDW, los operadores SQL de los datos HDFS se traducen en trabajos de MapReduce para su ejecución directa en el clúster Hadoop. Además, los datos HDFS se pueden importar y exportar a/desde PDW en paralelo, utilizando el mismo servicio de PDW que permite la redistribución de datos de PDW entre nodos de cómputo.

La arquitectura de Polybase, integrada en PDW, se muestra en la Fig. 11.11. Polybase aprovecha el Servicio de Movimiento de Datos (DMS) de PDW, responsable de la redistribución de datos intermedios entre los nodos de PDW, por ejemplo, para repartir tuplas, de modo que las tuplas coincidentes de una unión equi se ubiquen en el mismo nodo informático que realiza la unión. El DMS se amplía con un componente HDFS Bridge, responsable de todas las comunicaciones con HDFS.

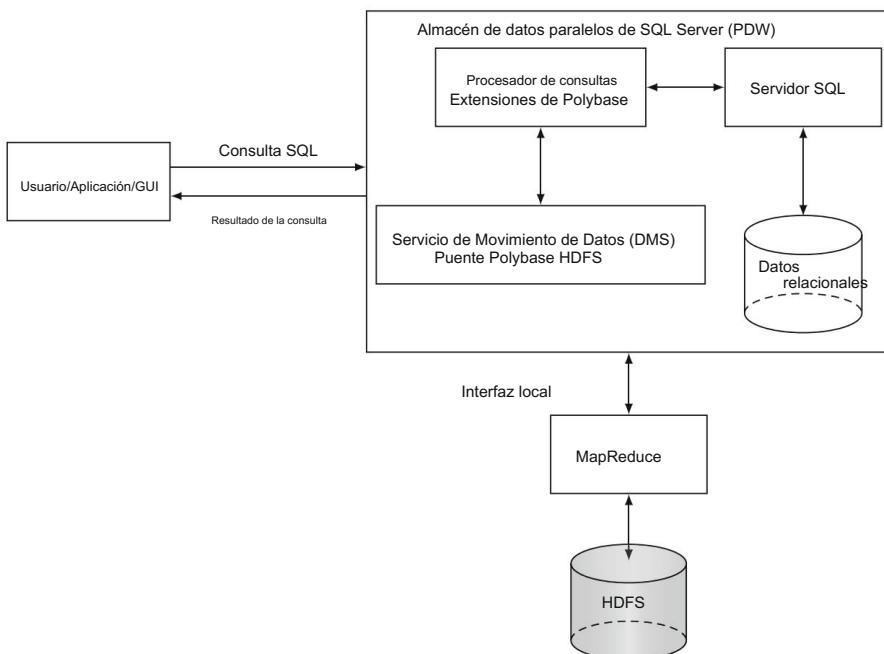


Fig. 11.11 Arquitectura de Polybase

HDFS Bridge permite que las instancias DMS también intercambien datos con HDFS en paralelo (accediendo directamente a las divisiones de HDFS).

Polybase utiliza el optimizador de consultas basado en costos de PDW para determinar cuándo es ventajoso transferir operaciones SQL sobre datos HDFS al clúster Hadoop para su ejecución. Por lo tanto, requiere estadísticas detalladas de las tablas externas, que se obtienen explorando muestras estadísticamente significativas de tablas HDFS. El optimizador de consultas enumera los QEP equivalentes y selecciona el de menor costo. El espacio de búsqueda se obtiene considerando las diferentes descomposiciones de la consulta en dos partes: una para ejecutarse como trabajos de MapReduce en el clúster Hadoop y la otra como operadores relationales regulares en PDW. Los trabajos de MapReduce pueden utilizarse para realizar operaciones de selección y proyección en tablas externas, así como uniones entre dos tablas externas. Los datos generados por los trabajos de MapReduce pueden exportarse a PDW para combinarse con datos relationales mediante algoritmos de unión paralelos basados en hash.

Una importante limitación de las operaciones de envío de datos HDFS como trabajos de MapReduce es que incluso las consultas de búsqueda simples presentan largas latencias. Una solución propuesta para Polybase consiste en explotar un índice generado a partir de los datos HDFS externos mediante un árbol B+ almacenado en PDW. Este método aprovecha el robusto y eficiente código de indexación de PDW sin forzar un aumento drástico del espacio necesario para almacenar o almacenar en caché todos los datos HDFS (grandes) dentro de PDW. Por lo tanto, el optimizador de consultas puede utilizar el índice como prefiltro para reducir la cantidad de trabajo que se realiza como trabajos de MapReduce. Para mantener el índice sincronizado con los datos almacenados en HDFS, se utiliza un enfoque incremental que registra que el índice está desactualizado y lo reconstruye de forma diferida. Las consultas realizadas al índice antes de que finalice el proceso de reconstrucción se pueden responder mediante un método que ejecuta cuidadosamente partes de la consulta utilizando el índice en PDW, y la parte restante se ejecuta como un trabajo de MapReduce únicamente sobre los datos modificados en HDFS. Apache AsterixDB utiliza un enfoque similar para acceder e indexar datos externos que residen en HDFS y permite que las consultas de los usuarios abarquen datos que AsterixDB administra, así como datos externos en HDFS.

11.7.2.2 HadoopDB

El objetivo de HadoopDB es ofrecer lo mejor de los sistemas de gestión de bases de datos (SGBD) paralelos (análisis de datos de alto rendimiento sobre datos estructurados) y basados en MapReduce (escalabilidad, tolerancia a fallos y flexibilidad para gestionar datos no estructurados) con un lenguaje similar a SQL (HiveQL) y un modelo de datos relacional. Para ello, HadoopDB integra estrechamente el framework de Hadoop, incluyendo MapReduce y HDFS, con múltiples SGBD relationales de un solo nodo implementados en un clúster, como en un SGBD paralelo sin recursos compartidos.

HadoopDB amplía la arquitectura de Hadoop con cuatro componentes: conector de base de datos, catálogo, cargador de datos y planificador SQL-MapReduce-SQL (SMS). El conector de base de datos proporciona los envoltorios al SGBD relacional subyacente mediante controladores JDBC. El catálogo mantiene información sobre las bases de datos como un archivo XML en HDFS y se utiliza para el procesamiento de consultas. El cargador de datos es responsable de

Reparticionamiento de colecciones de datos (clave, valor) mediante hash en una clave y carga de las particiones (o fragmentos) en bases de datos de un solo nodo. El planificador SMS extiende Hive, un componente de Hadoop que transforma HiveQL en trabajos de MapReduce que se conectan a tablas almacenadas como archivos en HDFS. Esta arquitectura genera un DBMS relacional paralelo rentable, donde los datos se partitionan tanto en tablas del DBMS relacional como en archivos HDFS, y las particiones se pueden ubicar en los nodos del clúster para un procesamiento paralelo eficiente.

El procesamiento de consultas es sencillo, y se basa en el planificador SMS para la traducción y optimización, y en MapReduce para la ejecución. La optimización consiste en transferir la mayor cantidad de trabajo posible a las bases de datos de un solo nodo y reparticionar las colecciones de datos cuando sea necesario. El planificador SMS descompone una consulta HiveQL en un QEP de operadores relativos. A continuación, los operadores se traducen a trabajos de MapReduce, mientras que los nodos hoja se transforman de nuevo en SQL para consultar las instancias del DBMS relacional subyacente. En MapReduce, el reparticionamiento debe realizarse antes de la fase de reducción. Sin embargo, si el optimizador detecta que una tabla de entrada está partitionada en una columna utilizada como clave de agregación para Reduce, simplifica el QEP convirtiéndolo en un único trabajo de Map, dejando que los nodos del DBMS relacional realicen toda la agregación. De igual forma, también se evita el reparticionamiento para las uniones de igualdad si ambos lados de la unión están partitionados en la clave de unión.

11.7.2.3 Estocada

Estocada es un polystore autoajustable cuyo objetivo es optimizar el rendimiento de las aplicaciones que gestionan datos en múltiples modelos, como relativos, clave-valor, de documento y de grafos. Para obtener el máximo rendimiento de los almacenes de datos disponibles, Estocada distribuye y partitiona automáticamente los datos entre los diferentes almacenes, que están completamente bajo su control y, por lo tanto, no son autónomos. Por lo tanto, es un polystore estrechamente acoplado.

La distribución de datos es dinámica y se decide mediante una combinación de heurística y decisiones basadas en costos, considerando los patrones de acceso a los datos a medida que están disponibles. Cada colección de datos se almacena como un conjunto de particiones, cuyo contenido puede solaparse, y cada partición puede almacenarse en cualquiera de los almacenes de datos subyacentes. Por lo tanto, es posible que una partición se almacene en un almacén de datos con un modelo de datos diferente al nativo. Para que las aplicaciones de Estocada sean independientes de los almacenes de datos, cada partición de datos se describe internamente como una vista materializada de una o varias colecciones de datos. Por lo tanto, el procesamiento de consultas implica la reescritura de consultas basada en vistas.

Estocada admite dos tipos de solicitudes, para almacenar datos y realizar consultas, con cuatro módulos principales: asesor de almacenamiento, catálogo, procesador de consultas y motor de ejecución. Estos componentes pueden acceder directamente a los almacenes de datos a través de su interfaz local. El procesador de consultas solo procesa consultas de un solo modelo, cada una expresada en el lenguaje de consulta de la fuente de datos correspondiente. Sin embargo, para integrar varias fuentes de datos, se necesitaría un modelo y lenguaje de datos comunes además de Estocada.

El asesor de almacenamiento es responsable de partitionar las recopilaciones de datos y delegar el almacenamiento de particiones a los almacenes de datos. Para optimizar las aplicaciones, puede...

También se recomienda reparticionar o mover datos de un almacén de datos a otro, según los patrones de acceso. Cada partición se define como una vista materializada, expresada como una consulta a la colección en su lenguaje nativo. El catálogo registra la información sobre las particiones, incluyendo información sobre el coste de las operaciones de acceso a los datos, mediante patrones de enlace específicos de los almacenes de datos.

Usando el catálogo, el procesador de consultas transforma una consulta sobre una colección de datos en un QEP lógico en, posiblemente, múltiples almacenes de datos (si existen particiones de la colección en diferentes almacenes). Esto se logra reescribiendo la consulta inicial utilizando las vistas materializadas asociadas a la colección de datos y seleccionando la mejor reescritura según los costos de ejecución estimados. El motor de ejecución convierte el QEP lógico en un QEP físico que puede ejecutarse directamente dividiendo el trabajo entre los almacenes de datos y el motor de ejecución de Estocada, que proporciona sus propios operadores (selección, unión, agregación, etc.).

11.7.3 Sistemas híbridos

Los sistemas híbridos buscan combinar las ventajas de los sistemas débilmente acoplados (p. ej., acceso a múltiples almacenes de datos diferentes) y los sistemas fuertemente acoplados (p. ej., acceso eficiente a algunos almacenes de datos directamente a través de sus interfaces locales). Por lo tanto, la arquitectura (véase la Fig. 11.12) sigue la arquitectura de mediador-encapsulador, mientras que el procesador de consultas también puede acceder directamente a algunos almacenes de datos, p. ej., HDFS, mediante MapReduce o Spark.

A continuación describimos tres polystores híbridos: Spark SQL, CloudMdsQL y BigDAWG.

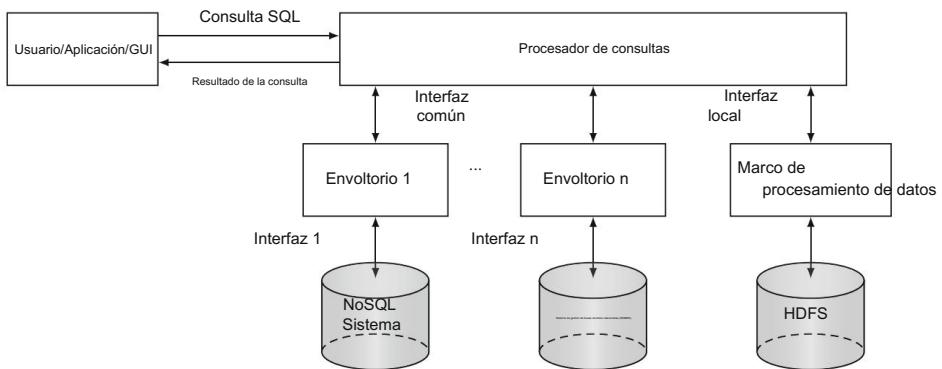


Fig. 11.12 Arquitectura de polialmacén híbrido

11.7.3.1 Spark SQL

Spark SQL es un módulo de Apache Spark que integra el procesamiento de datos relacionales con la API de programación funcional de Spark. Admite consultas tipo SQL que integran datos HDFS a los que se accede mediante Spark y fuentes de datos externas (p. ej., bases de datos relacionales) a las que se accede mediante un contenedor. Por lo tanto, es un polystore híbrido con un acoplamiento estrecho entre Spark y HDFS y un acoplamiento flexible con fuentes de datos externas.

Spark SQL cuenta con un modelo de datos relacional anidado. Admite los principales tipos de datos SQL, así como tipos definidos por el usuario y tipos de datos complejos (estructuras, matrices, mapas y uniones), que pueden anidarse. También admite DataFrames, que son conjuntos distribuidos de filas con el mismo esquema, como una tabla relacional.

Un DataFrame se puede construir a partir de una tabla en una fuente de datos externa o de un conjunto de datos distribuidos resilientes (RDD) de Spark existente con objetos nativos de Java o Python. Una vez construidos, los DataFrames se pueden manipular con diversos operadores relacionales, como WHERE y GROUPBY, que aceptan expresiones en código procedural de Spark.

La Figura 11.13 muestra la arquitectura de Spark SQL, que se ejecuta como una biblioteca sobre Spark. El procesador de consultas accede directamente al motor Spark a través de la interfaz Java de Spark, mientras que accede a fuentes de datos externas (p. ej., un SGBD relacional o un almacén de clave-valor) a través de la interfaz común de Spark SQL compatible con los controladores JDBC. El procesador de consultas incluye dos componentes principales: la API DataFrame y el optimizador de consultas Catalyst. La API DataFrame ofrece una estrecha integración entre el procesamiento relacional y procedimental, lo que permite realizar operaciones relacionales tanto en fuentes de datos externas como en RDD. Está integrada en los lenguajes de programación compatibles con Spark (Java, Scala, Python) y admite la ejecución en línea sencilla.

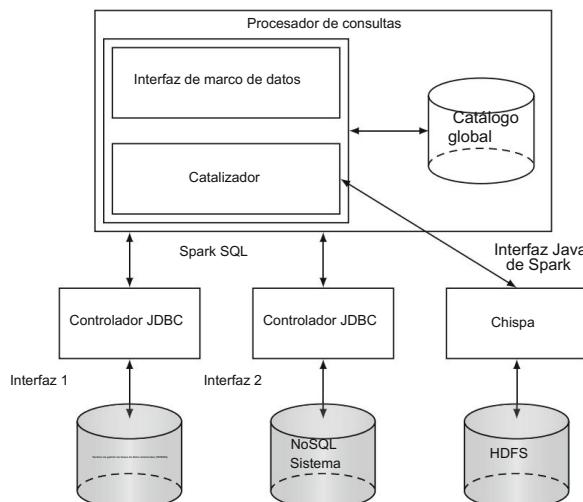


Figura 11.13 Arquitectura de Spark SQL

Definición de funciones definidas por el usuario, sin el complejo proceso de registro que suele encontrarse en otros sistemas de bases de datos. Por lo tanto, la API DataFrame permite a los desarrolladores combinar fluidamente la programación relacional y procedimental, por ejemplo, para realizar análisis avanzados (que resultan complejos de expresar en SQL) en grandes colecciones de datos (a las que se accede mediante operaciones relacionales).

Catalyst es un optimizador de consultas extensible que admite la optimización basada en reglas y costos. La motivación para un diseño extensible es facilitar la incorporación de nuevas técnicas de optimización, por ejemplo, para la compatibilidad con nuevas funciones de Spark SQL, así como para que los desarrolladores puedan ampliar el optimizador para gestionar fuentes de datos externas, por ejemplo, añadiendo reglas específicas de la fuente de datos para reducir los predicados de selección.

Aunque se han propuesto optimizadores de consultas extensibles en el pasado, estos suelen requerir un lenguaje complejo para especificar reglas y un compilador específico para traducirlas a código ejecutable. Por el contrario, Catalyst utiliza características estándar del lenguaje de programación funcional Scala, como la coincidencia de patrones, para facilitar a los desarrolladores la especificación de reglas, que pueden compilarse en código Java.

Catalyst proporciona un marco de transformación general para representar árboles de consulta y aplicar reglas para manipularlos. Este marco se utiliza en cuatro fases: (1) análisis de consultas, (2) optimización lógica, (3) optimización física y (4) generación de código. El análisis de consultas resuelve las referencias de nombres mediante un catálogo (con información del esquema) y genera un plan lógico. La optimización lógica aplica optimizaciones estándar basadas en reglas al plan lógico, como la inserción de predicados, la propagación de nulos y la simplificación de expresiones booleanas. La optimización física toma un plan lógico y enumera un espacio de búsqueda de planes físicos equivalentes, utilizando operadores físicos implementados en el motor de ejecución Spark o en las fuentes de datos externas. A continuación, selecciona un plan mediante un modelo de coste simple, en particular, para seleccionar los algoritmos de unión. La generación de código se basa en el lenguaje Scala, en particular, para facilitar la construcción de árboles de sintaxis abstracta (AST) en Scala. Los AST pueden entonces alimentarse al compilador de Scala en tiempo de ejecución para generar bytecode Java que será ejecutado directamente por los nodos de cómputo.

Para acelerar la ejecución de consultas, Spark SQL aprovecha el almacenamiento en caché en memoria de datos activos mediante un almacenamiento basado en columnas (es decir, almacena las colecciones de datos como secciones de columnas en lugar de filas). En comparación con la caché nativa de Spark, que simplemente almacena los datos como objetos nativos de Java, esta caché basada en columnas puede reducir el consumo de memoria significativamente mediante la aplicación de esquemas de compresión de columnas (p. ej., codificación de diccionario y codificación por longitud de ejecución). El almacenamiento en caché es especialmente útil para consultas interactivas y para los algoritmos iterativos comunes en el aprendizaje automático.

11.7.3.2 CloudMdsQL

CloudMdsQL admite un potente lenguaje funcional similar a SQL, diseñado para consultar múltiples fuentes de datos heterogéneas (p. ej., relacionales y NoSQL). Una consulta de CloudMdsQL puede contener subconsultas anidadas, cada una de las cuales se dirige directamente a un almacén de datos específico y puede contener invocaciones integradas a la interfaz de consulta nativa del almacén de datos. Por lo tanto, la principal innovación radica en que una consulta de CloudMdsQL...

Puede aprovechar al máximo el potencial de los almacenes de datos locales, simplemente permitiendo que algunas consultas nativas de estos almacenes (por ejemplo, una consulta de búsqueda en amplitud en una base de datos de grafos) se invoquen como funciones. CloudMdsQL se ha ampliado para adaptarse a marcos de procesamiento distribuido como Apache Spark, permitiendo el uso ad hoc de operadores de mapa, filtro y reducción definidos por el usuario como subconsultas.

El lenguaje CloudMdsQL se basa en SQL y ofrece capacidades ampliadas para integrar subconsultas expresadas en la interfaz de consulta nativa de cada almacén de datos. El modelo de datos común se basa en tablas y admite tipos de datos enriquecidos que capturan una amplia gama de tipos de datos subyacentes del almacén de datos, como matrices y objetos JSON, para gestionar datos no planos y anidados, con operadores básicos sobre estos tipos de datos compuestos. CloudMdsQL permite definir expresiones de tabla con nombre como funciones de Python, lo cual resulta útil para consultar almacenes de datos que solo tienen una interfaz de consulta basada en API. Una consulta de CloudMdsQL se ejecuta en el contexto de un esquema ad hoc, formado por todas las expresiones de tabla con nombre dentro de la consulta. Este enfoque cubre la falta de un esquema global y permite al compilador de consultas realizar un análisis semántico de la consulta.

El diseño del motor de consultas CloudMdsQL aprovecha su funcionamiento en una plataforma en la nube, con control total sobre la ubicación de los componentes del sistema. La arquitectura del motor de consultas es completamente distribuida, de modo que sus nodos pueden comunicarse directamente entre sí mediante el intercambio de código (planes de consulta) y datos. Esta arquitectura distribuida ofrece importantes oportunidades de optimización, por ejemplo, minimizando las transferencias de datos al transferir los datos intermedios más pequeños para su posterior procesamiento por un nodo específico. Cada nodo del motor de consultas consta de dos partes (maestro y trabajador) y se ubica en cada nodo de almacenamiento de datos de un clúster. Cada maestro o trabajador cuenta con un procesador de comunicación que admite operadores de envío y recepción para intercambiar datos y comandos entre nodos.

Un maestro toma como entrada una consulta y, mediante un planificador y catálogo de consultas (con metadatos e información de costes de las fuentes de datos), genera un plan de consulta que envía a un nodo del motor de consultas seleccionado para su ejecución. Cada trabajador actúa como un procesador de base de datos ligero en tiempo de ejecución sobre un almacén de datos y se compone de tres módulos genéricos (es decir, la misma biblioteca de código): controlador de ejecución de consultas, motor de operadores y almacenamiento de tablas, y un módulo contenedor específico para cada almacén de datos.

El planificador de consultas realiza una optimización basada en costes. Para comparar alternativas de reescritura de una consulta, el optimizador utiliza un catálogo simple que proporciona información básica sobre las colecciones del almacén de datos, como cardinalidades, selectividades de atributos e índices, y un modelo de costes simple. Los envoltorios pueden exponer dicha información en forma de funciones de coste o estadísticas de la base de datos. El lenguaje de consulta también permite al usuario definir funciones de coste y selectividad cuando no se pueden derivar del catálogo, principalmente al utilizar subconsultas nativas. El espacio de búsqueda de planes alternativos se obtiene mediante transformaciones tradicionales, por ejemplo, mediante la inserción de predicados de selección, el uso de uniones de enlace, la ordenación de uniones o la planificación del envío de datos intermedios.

11.7.3.3 Gran DAWG

Al igual que los sistemas multibase de datos, todos los polystores que hemos visto hasta ahora proporcionan acceso transparente a través de múltiples almacenes de datos con el mismo modelo de datos y lenguaje. BigDAWG (Big Data Analytics Working Group) adopta un enfoque diferente, con el objetivo de unificar las consultas en diversos modelos y lenguajes de datos. Por lo tanto, no existe un modelo ni lenguaje de datos común. Una abstracción clave para el usuario en BigDAWG es una isla de información, que consiste en una colección de almacenes de datos a los que se accede con un único lenguaje de consulta. Puede haber diversas islas, incluyendo sistemas de gestión de bases de datos relacionales, sistemas de gestión de bases de datos de matriz, NoSQL y sistemas de flujo de datos (DSS). Dentro de una isla, existe un acoplamiento flexible de los almacenes de datos, que deben proporcionar un contenedor (denominado shim) para asignar el lenguaje de la isla a su lenguaje nativo. Cuando una consulta accede a más de un almacén de datos, es posible que sea necesario copiar objetos entre bases de datos locales mediante una operación CAST, que proporciona un tipo de acoplamiento estrecho. Por ello, BigDAWG puede considerarse un polialmacén híbrido.

La arquitectura de BigDAWG es altamente distribuida, con una capa delgada que conecta las herramientas (p. ej., visualización) y las aplicaciones con las islas de información. Al no existir un modelo de datos ni un lenguaje comunes, tampoco existe un procesador de consultas común. En su lugar, cada isla tiene su propio procesador de consultas.

El procesamiento de consultas dentro de una isla es similar al de los sistemas multibase de datos: la mayor parte del procesamiento se envía a los almacenes de datos y el procesador de consultas solo integra los resultados. El optimizador de consultas no utiliza un modelo de costes, sino heurísticas y cierto conocimiento del alto rendimiento de algunos almacenes de datos. Para consultas sencillas, como select-project-join, el optimizador utilizará el envío de funciones para minimizar el movimiento de datos y el tráfico de red entre los almacenes de datos. Para consultas complejas, como las analíticas, el optimizador puede considerar el envío de datos para trasladarlos a un almacén de datos que ofrezca una implementación de alto rendimiento.

Una consulta enviada a una isla puede involucrar varias islas. En este caso, la consulta debe expresarse como varias subconsultas, cada una en el idioma de una isla específica.

Para especificar la isla a la que se destina una subconsulta, el usuario la encierra en una especificación SCOPE. Por lo tanto, una consulta multiisla tendrá múltiples ámbitos para indicar el comportamiento esperado de sus subconsultas. Además, el usuario puede insertar operaciones CAST para mover conjuntos de datos intermedios entre islas de forma eficiente. Por lo tanto, el procesamiento de consultas multiisla depende de cómo el usuario especifique las subconsultas, las operaciones SCOPE y CAST.

11.7.4 Observaciones finales

Aunque todos los polystores comparten el mismo objetivo general de consultar múltiples almacenes de datos, existen muchas maneras diferentes de lograrlo, dependiendo del objetivo funcional que se desee alcanzar. Este objetivo tiene un impacto importante en las decisiones de diseño. La principal tendencia predominante es la capacidad de integrar datos relacionales (almacenados en SGBD relacionales) con otros tipos de datos en diferentes almacenes de datos, como

como HDFS (Polybase, HadoopDB, Spark SQL, JEN) o NoSQL (Bigtable solo para BigIntegrator, almacenes de documentos para Forward). Por lo tanto, una diferencia importante radica en los tipos de almacenes de datos compatibles. Por ejemplo, Estocada, BigDAWG y CloudMdsQL admiten una amplia variedad de almacenes de datos, mientras que Polybase y JEN se centran en la integración de sistemas de gestión de bases de datos relacionales (SGBD) exclusivamente con HDFS. Cabe destacar también la creciente importancia del acceso a HDFS dentro de Hadoop, en particular con MapReduce o Spark, lo que se corresponde con los principales casos de uso en la integración de datos estructurados y no estructurados.

Otra tendencia es el surgimiento de polialmacenes autoajustables, como Estocada y Odyssey, con el objetivo de aprovechar los almacenes de datos disponibles para mejorar el rendimiento.

En cuanto al modelo de datos y el lenguaje de consulta, la mayoría de los sistemas ofrecen una abstracción relacional similar a SQL. Sin embargo, QoX cuenta con una abstracción gráfica más general para capturar flujos de datos analíticos. Tanto Estocada como BigDAWG permiten acceder directamente a los almacenes de datos con sus lenguajes nativos (o lenguajes de isla). CloudMdsQL también permite consultas nativas, pero como subconsultas dentro de un lenguaje similar a SQL.

La mayoría de los polystores ofrecen compatibilidad con la gestión de esquemas globales, utilizando los enfoques GAV o LAV, con algunas variaciones. Por ejemplo, BigDAWG utiliza GAV dentro de islas de información (de un solo modelo). Sin embargo, QoX, Estocada, Spark SQL y CloudMdsQL no admiten esquemas globales, aunque ofrecen una forma de gestionar los esquemas locales de los almacenes de datos.

Las técnicas de procesamiento de consultas son extensiones de técnicas conocidas de sistemas de bases de datos distribuidas, como el envío de datos/funciones, la descomposición de consultas (basada en las capacidades de los almacenes de datos), la unión de enlaces y la selección de pushdown. La optimización de consultas también suele ser compatible, ya sea con un modelo de coste (simple) o heurística.

11.8 Conclusión

En comparación con los SGBD relacionales tradicionales, estas nuevas tecnologías prometen mayor escalabilidad, rendimiento y facilidad de uso. Además, complementan las nuevas tecnologías de gestión de datos para big data (véase el capítulo 10).

La principal motivación de NoSQL es abordar tres limitaciones principales de los SGBD relacionales: un enfoque universal para todo tipo de datos y aplicaciones; la limitada escalabilidad y disponibilidad de la arquitectura de base de datos en la nube; y, como lo demuestra el teorema CAP, la disyuntiva entre una sólida consistencia de la base de datos y la disponibilidad del servicio. Las cuatro categorías principales de sistemas NoSQL se basan en su modelo de datos subyacente: clave-valor, columna ancha, documento y grafo.

Para cada categoría, se ilustró con un sistema representativo: DynamoDB (clave-valor), Bigtable (columna ancha), MongoDB (documento) y Neo4j (grafo). También se ilustraron sistemas NoSQL multimodelo con OrientDB, que combina conceptos de modelos de datos de documentos y grafos orientados a objetos y NoSQL.

Los sistemas NoSQL proporcionan escalabilidad, así como disponibilidad, esquemas flexibles y API prácticas, pero esto generalmente se logra relajando la consistencia estricta de la base de datos. NewSQL es una clase reciente de SGBD que busca combinar...

Escalabilidad de los sistemas NoSQL con la alta consistencia y usabilidad de los SGBD relacionales. El objetivo es abordar los requisitos de los sistemas de información empresarial, que han sido respaldados por los SGBD relacionales tradicionales, pero que también necesitan escalabilidad. Ilustramos NewSQL con los SGBD Google F1 y LeanXcale.

La creación de aplicaciones intensivas en datos en la nube a menudo requiere el uso de múltiples almacenes de datos (NoSQL, HDFS, DBMS relacionales, NewSQL), cada uno optimizado para un tipo de datos y tareas. En particular, muchos casos de uso exhiben la necesidad de combinar datos de estructura flexible (p. ej., archivos de registro, tweets, páginas web) que se admiten mejor en HDFS o NoSQL con datos más estructurados en DBMS relacionales. Los polystores proporcionan acceso integrado o transparente a varios almacenes de datos en la nube a través de uno o más lenguajes de consulta. Dividimos los polystores en función del nivel de acoplamiento con los almacenes de datos subyacentes, es decir, acoplados de forma flexible, acoplados de forma estrecha e híbridos. Luego, presentamos tres polystores representativos para cada clase: BigIntegrator, Forward y QoX (acoplados de forma flexible); Polybase, HadoopDB y Estocada (acoplados de forma estrecha); Spark SQL, CloudMdsQL y BigDAWG (híbridos).

La principal tendencia predominante es la capacidad de integrar datos relacionales (almacenados en sistemas de gestión de bases de datos relacionales) con otros tipos de datos en diferentes almacenes de datos, como HDFS o NoSQL. Sin embargo, una diferencia importante entre los polialmacenes reside en el tipo de almacenes de datos que admiten. También destacamos la creciente importancia del acceso a HDFS dentro de Hadoop, en particular con plataformas de procesamiento de big data como MapReduce o Spark. Otra tendencia es la aparición de polialmacenes autoajustables, con el objetivo de optimizar el rendimiento de los almacenes de datos disponibles. En cuanto al modelo de datos y el lenguaje de consulta, la mayoría de los sistemas ofrecen una abstracción relacional/similar a SQL. Sin embargo, QoX tiene una abstracción gráfica más general para capturar flujos de datos analíticos. Tanto Estocada como BigDAWG permiten acceder directamente a los almacenes de datos con sus lenguajes nativos. Las técnicas de procesamiento de consultas son extensiones de técnicas conocidas de los sistemas de bases de datos distribuidas (véase el capítulo 4).

11.9 Notas bibliográficas

El panorama en NoSQL, NewSQL y polystores cambia constantemente y carece de estándares, lo que dificulta la elaboración de una bibliografía buena y actualizada.

Existen numerosos libros y artículos de investigación sobre el tema, pero se desactualizan rápidamente. Se puede encontrar información adicional y actualizada en los sitios web y blogs de sistemas. En este capítulo, nos centramos en los principios y arquitecturas de los sistemas, en lugar de en los detalles de implementación que pueden cambiar con el tiempo.

Una motivación frecuentemente citada para NoSQL es el teorema CAP, que ayuda a comprender el equilibrio entre (C) consistencia, (A) disponibilidad y (P) tolerancia a particiones. Comenzó como una conjectura de [Brewer 2000] y se convirtió en teorema de [Gilbert y Lynch 2002]. Aunque el teorema CAP no menciona la escalabilidad, algunos NoSQL lo han utilizado para justificar la falta de compatibilidad con transacciones ACID.

Existen varios libros que introducen el movimiento NoSQL, en particular [Strauch 2011, Redmond y Wilson 2012], que ilustran bien el tema con varios sistemas representativos presentados en este capítulo. También existen buenos libros sobre sistemas específicos. La presentación del almacén de documentos de MongoDB se basa en el libro [Plugge et al. 2010] y otra información disponible en el sitio web de MongoDB. AsterixDB [Alsubaiee et al. 2014] y Couchbase [Borkar et al. 2016] son almacenes de documentos JSON que admiten un dialecto de SQL++, propuesto inicialmente en [Ong et al. 2014]. También existe un excelente libro práctico sobre SQL++ [Chamberlin 2018] escrito por Don Chamberlin, coinventor del lenguaje SQL original. El mecanismo de acceso e indexación de datos externos de AsterixDB se encuentra en [Alamoudi et al. 2015]. También existen buenas descripciones de DynamoDB [DeCandia et al., 2007] y Bigtable [Chang et al., 2008]. Para una introducción a las bases de datos gráficas y Neo4j, se puede consultar el excelente libro del equipo de Neo4j [Robinson et al., 2015]. Neo4j utiliza el modelo de consistencia causal [Elbushra y Lindström, 2015] para la replicación multimaestro y el protocolo Raft para la durabilidad de las transacciones [Ongaro y Ousterhout, 2014].

La sección sobre sistemas NewSQL se basa en la descripción del DBMS F1 [Shute et al. 2013] y el DBMS LeanXcale HTAP [Jimenez-Peris y Patiño Martínez 2011, Kolev et al. 2018].

Una buena motivación para los polialmacenes, o sistemas multialmacenamiento, se puede encontrar en [Duggan et al. 2015, Kolev et al. 2016b]. La sección sobre polialmacenes se basa en nuestro artículo de investigación sobre el procesamiento de consultas en sistemas multialmacenamiento [Bondiombou y Valduriez 2016]. Este artículo identifica tres clases de sistemas (1) débilmente acoplados, (2) fuertemente acoplados y (3) híbridos e ilustra cada clase con tres sistemas representativos: (1) BigIntegrator [Zhu y Risch 2011], Forward [Fu et al. 2014] y QoS [Simitsis et al. 2009, 2012]; (2) Polybase [DeWitt et al. 2013, Gankidi et al. 2014], HadoopDB [Abouzeid et al. 2009] y Estocada [Bugiotti et al. 2015]; (3) Spark SQL [Armbrust et al. 2015], BigDAWG [Gadepally et al. 2016] y CloudMdsQL [Bondiombouy et al. 2016, Kolev et al. 2016b,a]. Otros polialmacenes importantes son Amazon Redshift Spectrum, AsterixDB y AWESOME [Dasgupta et al. 2016], Odyssey [Hacigümüs et al. 2013] y JEN [Tian et al. 2016]. Odyssey utiliza vistas materializadas oportunistas, basadas en MISO [LeFevre et al. 2014], un método para ajustar el diseño físico de un polialmacén.

Ceremonias

Problema 11.1 Recuerde y analice las motivaciones para NoSQL, en particular en comparación con los DBMS relacionales.

Problema 11.2 (*) Explique la importancia del teorema CAP. Considere una arquitectura distribuida con replicación multimaestro (véase el capítulo 6). Suponga una red particionada con replicación asíncrona.

- (a) ¿Cuáles de las propiedades del CAP se conservan? (b) ¿Qué tipo de consistencia se logra?

Las mismas preguntas asumiendo replicación sincrónica.

Problema 11.3 (***) En este capítulo, dividimos los sistemas NoSQL en cuatro categorías, es decir, clave-valor, columna ancha, documento y gráfico.

- (a) Analice las principales similitudes y diferencias en términos de modelo de datos, lenguaje de consulta e interfaces, arquitecturas y técnicas de implementación.
- (b) Identifique los mejores casos de uso para cada categoría de sistema.

Problema 11.4 (**) Considere el siguiente esquema de base de datos de ingreso de pedidos simplificado (en formato relacional anidado), con atributos de clave principal subrayados:

```

CLIENTES (CID, NOMBRE, DIRECCIÓN (CALLE, CIUDAD, ESTADO,
PAÍS), TELÉFONOS)
PEDIDOS(OID, CID, O-FECHA, O-TOTAL)
ARTÍCULOS DE PEDIDO(OID, ID DE LÍNEA, PID, CANTIDAD)
PRODUCTOS(PID, P-NOMBRE, PRECIO)

```

- (a) Indique los esquemas correspondientes en los cuatro tipos de sistemas NoSQL (clave-valor, columna ancha, documento y grafo). Analice las ventajas y desventajas de cada diseño en términos de facilidad de uso, administración de bases de datos, complejidad de consultas y rendimiento de actualización. (b) Considere ahora que un producto puede estar compuesto por varios productos, por ejemplo, un paquete de seis cervezas. Reflexione sobre esto en los esquemas de bases de datos y analice las implicaciones utilizando los cuatro tipos de sistemas NoSQL.

Problema 11.5 (**) Como se explicó en el capítulo 10, no existe una solución óptima para la partición de bases de datos gráficas. Explique el impacto en la escalabilidad de las bases de datos gráficas.

Proporcionar formas de solucionarlo.

Problema 11.6 (**) Compare el sistema F1 NewSQL con un DBMS relacional paralelo estándar, por ejemplo, MySQL Cluster, en términos de modelo de datos, lenguaje de consulta e interfaces, consistencia, escalabilidad y disponibilidad.

Problema 11.7 Los polystores proporcionan acceso integrado mediante consultas a múltiples almacenes de datos, como NoSQL, sistemas de gestión de bases de datos relacionales o HDFS. Compare los polystores con los sistemas de integración de datos que presentamos en el capítulo 7.

Problema 11.8 (***)) Los polystores suelen admitir consultas de solo lectura, lo que satisface los requisitos de análisis. Sin embargo, a medida que se crean sistemas de almacenamiento en la nube cada vez más complejos y con un uso intensivo de datos, la necesidad de actualizar los datos entre los almacenes de datos se volverá más importante. Por lo tanto, surgirá la necesidad de transacciones distribuidas. No obstante, los modelos de transacción de los almacenes de datos pueden ser muy diferentes. En particular, la mayoría de los sistemas NoSQL no admiten transacciones ACID. Analice el problema y proponga soluciones.

Capítulo 12

Gestión de datos web



La World Wide Web («WWW» o «web» para abreviar) se ha convertido en un importante repositorio de datos y documentos. Si bien las mediciones difieren y cambian, la web ha crecido a un ritmo vertiginoso.¹ Además de su tamaño, la web es muy dinámica y cambia rápidamente. A efectos prácticos, la web representa un almacén de datos muy grande, dinámico y distribuido, y existen los evidentes problemas de gestión de datos distribuidos al acceder a ellos.

La web, en su forma actual, puede considerarse como dos componentes distintos pero relacionados. El primero es la web públicamente indexable (PIW), compuesta por todas las páginas web estáticas (y con enlaces cruzados) que existen en servidores web. Estas se pueden buscar e indexar fácilmente. El otro componente, conocido como la web profunda (o web oculta), está compuesto por una gran cantidad de bases de datos que encapsulan los datos, ocultándolos del mundo exterior. Los datos de la web oculta se recuperan generalmente mediante interfaces de búsqueda donde el usuario introduce una consulta que se envía al servidor de bases de datos y los resultados se devuelven al usuario como una página web generada dinámicamente. Una parte de la web profunda se conoce como la «web oscura», que consiste en datos cifrados y requiere un navegador específico como Tor para acceder a ella.

La diferencia entre la PIW y la web oculta radica básicamente en cómo se gestionan para las búsquedas y consultas. La búsqueda en la PIW se basa principalmente en el rastreo de sus páginas mediante la estructura de enlaces entre ellas, la indexación de las páginas rastreadas y la posterior búsqueda de los datos indexados (como se explica en detalle en la sección 12.2).

Esto puede realizarse mediante la conocida búsqueda por palabras clave o mediante sistemas de preguntas y respuestas (QA) (Sección 12.4). No es posible aplicar este enfoque directamente a la web oculta, ya que no es posible rastrear e indexar esos datos (las técnicas de búsqueda en la web oculta se describen en la Sección 12.5).

La investigación sobre la gestión de datos web ha seguido diferentes líneas en dos comunidades separadas pero superpuestas. La mayor parte del trabajo anterior en la búsqueda web y

¹Véase <http://www.worldwidewebsize.com/>.

Comunidad de recuperación de información centrada en la búsqueda de palabras clave y motores de búsqueda. El trabajo posterior en esta comunidad se centró en los sistemas de control de calidad. El trabajo en la comunidad de bases de datos se centró en las consultas declarativas de datos web. Existe una tendencia emergente que combina el acceso mediante búsqueda/navegación con las consultas declarativas, pero este trabajo aún no ha alcanzado su máximo potencial. En la década del 2000, XML surgió como un formato de datos importante para la representación e integración de datos en la web. Por lo tanto, la gestión de datos XML fue un tema de gran interés. Si bien XML sigue siendo importante en diversas áreas de aplicación, su uso en la gestión de datos web ha disminuido, principalmente debido a su aparente complejidad. Más recientemente, RDF se ha consolidado como una representación común para la representación e integración de datos web.

El resultado de estos diferentes hilos de desarrollo es que hay poco en cuanto a una arquitectura o marco unificador para discutir la gestión de datos web, y las diferentes líneas de investigación deben considerarse de forma algo separada.

Además, la cobertura completa de todos los temas relacionados con la web requiere un tratamiento mucho más profundo y extenso del que se puede incluir en un solo capítulo. Por lo tanto, nos centramos en temas directamente relacionados con la gestión de datos.

Comenzamos analizando cómo se pueden modelar los datos web como un grafo. Tanto la estructura de este grafo como su gestión son importantes. Esto se aborda en la Sección 12.1. La búsqueda web se aborda en la Sección 12.2 y las consultas web se abordan en la Sección 12.3. La Sección 12.4 resume los sistemas de preguntas y respuestas, y la búsqueda y consulta en la web profunda/oculta se aborda en la Sección 12.5. Posteriormente, en la Sección 12.6, analizamos la integración de datos web, centrándonos tanto en los problemas fundamentales como en algunos enfoques de representación (p. ej., tablas web, XML y RDF) que pueden facilitar esta tarea.

12.1 Gestión de gráficos web

La web se compone de páginas conectadas mediante hipervínculos, y esta estructura puede modelarse como un grafo dirigido que refleja la estructura del hipervínculo. En este grafo, comúnmente conocido como grafo web, las páginas web HTML estáticas son los vértices y los enlaces entre ellas se representan como aristas dirigidas. Las características del grafo web son importantes para el estudio de la gestión de datos, ya que su estructura se utiliza en la búsqueda web, la categorización y clasificación de contenido web y otras tareas relacionadas con la web. Además, la representación RDF, que se describe en la sección 12.6.2.2, formaliza el grafo web mediante una notación específica. Las características importantes del grafo web son las siguientes:

- (a) Es bastante volátil. Ya hemos comentado la velocidad de crecimiento del grafo. Además, una proporción significativa de las páginas web se actualizan con frecuencia.
- (b) Es disperso. Un grafo se considera disperso si su grado promedio (es decir, el promedio de los grados de todos sus vértices) es menor que el número de vértices. Esto

Significa que cada vértice del grafo tiene un número limitado de vecinos, incluso si los vértices están generalmente conexos. La escasez del grafo de red implica una estructura gráfica interesante que analizaremos en breve.

- (c) Es autoorganizada. La web contiene varias comunidades, cada una de las cuales consta de un conjunto de páginas centradas en un tema específico. Estas comunidades se organizan de forma autónoma, sin ningún control centralizado, y dan lugar a los subgrafos particulares del grafo web.
- (d) Es un "grafo de mundo pequeño". Esta propiedad está relacionada con la escasez: cada nodo en el grafo puede no tener muchos vecinos (es decir, su grado puede ser pequeño), pero muchos nodos están conectados a través de intermediarios. Las redes de mundo pequeño se identificaron por primera vez en las ciencias sociales, donde se observó que muchas personas que son desconocidas entre sí están conectadas por intermediarios. Esto también es cierto en los grafos de red en términos de la conectividad del grafo. (e) Es un grafo de ley de potencia. Las distribuciones de grado de entrada y salida del grafo de red siguen distribuciones de ley de potencia. Esto significa que la probabilidad de que un vértice tenga grado de entrada (salida) i es proporcional a $1/i^\alpha$ para algún $\alpha > 1$. El valor de α es aproximadamente 2,1 para grado de entrada y aproximadamente 7,2 para grado de salida.

Esto nos lleva a analizar la estructura del grafo web, que tiene forma de "pajarita" (Fig. 12.1).

Tiene un componente fuertemente conectado (el nudo central) en el que existe una ruta entre cada par de páginas. Las cifras que presentamos a continuación provienen de un estudio realizado en el año 2000; si bien es posible que estas cifras hayan cambiado, la estructura representada en la figura se ha mantenido. Los lectores deben considerar las cifras como un indicador de tamaño relativo y no como valores absolutos. El componente fuertemente conectado (SCC) representa aproximadamente el 28 % de las páginas web. Otro 21 % de las páginas constituye el componente "IN", desde el cual existen rutas a páginas en SCC, pero no existen rutas desde páginas en SCC. Simétricamente, el componente "OUT" tiene páginas a las que existen rutas desde páginas en SCC, pero no viceversa, y estas también constituyen el 21 % de las páginas. Los "zarcillos" consisten en páginas a las que no se puede acceder desde SCC y desde las cuales no se puede acceder a páginas en SCC.

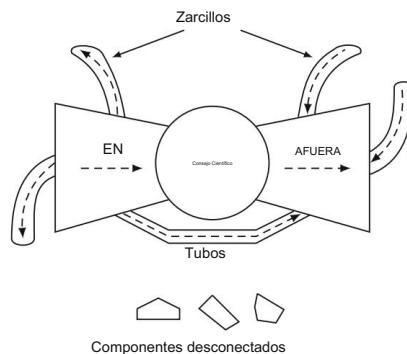


Fig. 12.1 La estructura de la red como una pajarita (basado en [Kumar et al. 2000])

Estos constituyen aproximadamente el 22% de las páginas web. Son páginas que aún no se han "descubierto" ni se han conectado a las partes más conocidas de la web. Finalmente, hay componentes desconectados que no tienen enlaces a nada más que a sus propias pequeñas comunidades. Esto representa aproximadamente el 8% de la web.

Esta estructura es interesante porque determina los resultados que se obtienen de las búsquedas y consultas web. Además, esta estructura gráfica es diferente a la de muchos otros gráficos que se estudian habitualmente, lo que requiere algoritmos y técnicas especiales para su gestión.

12.2 Búsqueda web

La búsqueda web implica encontrar todas las páginas web relevantes (es decir, con contenido relacionado) para las palabras clave que especifica el usuario. Naturalmente, no es posible encontrar todas las páginas, ni siquiera saber si se han recuperado todas; por lo tanto, la búsqueda se realiza en una base de datos de páginas web recopiladas e indexadas. Dado que suele haber varias páginas relevantes para una consulta, estas se presentan al usuario en orden de relevancia, según lo determine el motor de búsqueda.

La arquitectura abstracta de un motor de búsqueda genérico se muestra en la figura 12.2. Analizamos los componentes de esta arquitectura con cierto detalle.

En todo motor de búsqueda, el rastreador desempeña una de las funciones más cruciales. Un rastreador es un programa que utiliza un motor de búsqueda para escanear la web en su nombre y recopilar datos sobre las páginas web. Al rastreador se le asigna un conjunto inicial de páginas; más precisamente, se le asigna un conjunto de Localizadores Uniformes de Recursos (URL) que identifican estas páginas. El rastreador recupera y analiza la página correspondiente a esa URL, extrae cualquier

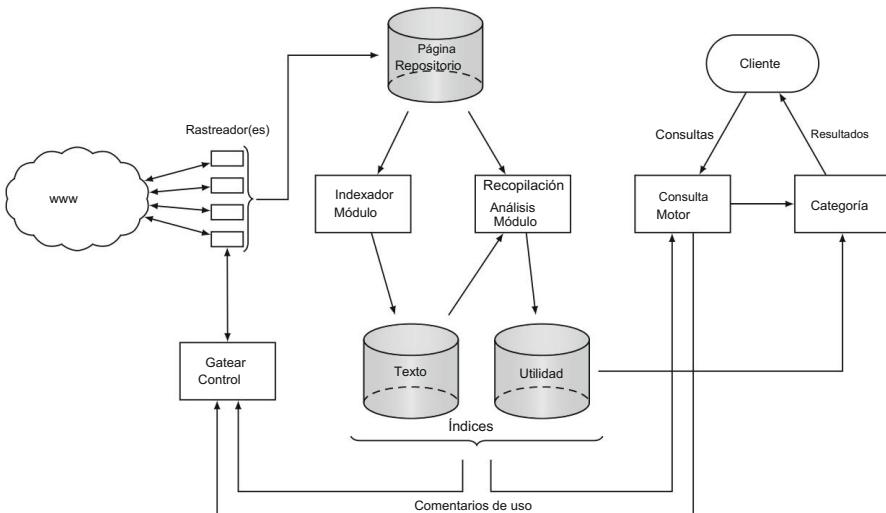


Fig. 12.2 Arquitectura del motor de búsqueda (basado en [Arasu et al. 2001])

Las URL que contiene y las añade a una cola. En el siguiente ciclo, el rastreador extrae una URL de la cola (según un orden) y recupera la página correspondiente.

Este proceso se repite hasta que el rastreador se detiene. Un módulo de control se encarga de decidir qué URL se deben visitar a continuación. Las páginas recuperadas se almacenan en un repositorio de páginas. La sección [12.2.1](#) analiza las operaciones de rastreo con más detalle.

El módulo indexador se encarga de construir índices en las páginas descargadas por el rastreador. Si bien se pueden construir muchos índices diferentes, los dos más comunes son los de texto y los de enlaces. Para construir un índice de texto, el módulo indexador crea una gran "tabla de búsqueda" que proporciona todas las URL que apuntan a las páginas donde aparece una palabra determinada. Un índice de enlaces describe la estructura de enlaces de la web y proporciona información sobre el estado de los enlaces entrantes y salientes de las páginas. La sección [12.2.2](#) explica la tecnología de indexación actual y se centra en cómo almacenar los índices de forma eficiente.

El módulo de clasificación se encarga de ordenar un gran número de resultados, presentando primero aquellos considerados más relevantes para la búsqueda del usuario. El problema de la clasificación ha suscitado un creciente interés para ir más allá de las técnicas tradicionales de recuperación de información (IR) y abordar las características especiales de la web: las consultas web suelen ser pequeñas y se ejecutan sobre una gran cantidad de datos. La Sección [12.2.3](#) presenta algoritmos de clasificación y describe enfoques que aprovechan la estructura de enlaces de la web para obtener mejores resultados de clasificación.

12.2.1 Rastreo web

Como se indicó anteriormente, un rastreador web escanea la web en nombre de un motor de búsqueda para extraer información sobre las páginas web visitadas. Dado el tamaño de la web, la naturaleza cambiante de las páginas web y las limitadas capacidades de procesamiento y almacenamiento de los rastreadores, es imposible rastrear toda la web. Por lo tanto, un rastreador debe estar diseñado para visitar las páginas "más importantes" antes que otras. La cuestión, entonces, es visitar las páginas en orden de importancia.

Hay una serie de cuestiones que deben abordarse al diseñar un rastreador.

Dado que el objetivo principal es acceder a las páginas más importantes antes que a otras, es necesario determinar la importancia de una página. Esto se puede lograr mediante una medida que refleje la importancia de una página determinada. Estas medidas pueden ser estáticas, de modo que la importancia de una página se determina independientemente de las consultas de recuperación que se ejecuten en ella, o dinámicas, ya que tienen en cuenta las consultas. Ejemplos de medidas estáticas son las que determinan la importancia de P_i de una página con respecto al número de páginas que apuntan a P_i (denominadas backlinks), o las que, además, consideran la importancia de las páginas de backlinks, como se hace en la popular métrica PageRank, utilizada por Google y otros.

Una posible medida dinámica puede ser una que calcule la importancia de una página P_i con respecto a su similitud textual con la consulta que se está evaluando utilizando algunas de las medidas de similitud de recuperación de información bien conocidas.

Introdujimos el PageRank en el capítulo 10 (ejemplo 10.4). Recordemos que el PageRank de una página P_i , denotado como $PR(P_i)$, es simplemente la suma normalizada del PageRank de todas las páginas de backlinks de P_i (denotadas como B_Pi) , donde la normalización para cada $P_j \in B_Pi$ se aplica a todos los enlaces directos de P_j a P_i :

$$PR(P_i) = \frac{\sum_{P_j \in B_Pi} PR(P_j)}{|B_Pi|}$$

Recuerde también que esta fórmula calcula el ranking de una página según los backlinks, pero normaliza la contribución de cada página que enlaza (P_j) utilizando el número de enlaces directos que tiene . La idea es que es más importante ser enlazado por páginas que enlazan de forma conservadora a otras páginas que por aquellas que enlazan a otras indiscriminadamente. Sin embargo, la contribución de un enlace desde dicha página debe normalizarse para todas las páginas a las que apunta.

Un segundo problema es cómo el rastreador elige la siguiente página a visitar una vez que ha rastreado una página en particular. Como se mencionó anteriormente, el rastreador mantiene una cola donde almacena las URL de las páginas que descubre al analizar cada una. Por lo tanto, el problema radica en ordenar las URL en esta cola. Existen varias estrategias posibles. Una posibilidad es visitar las URL en el orden en que se descubrieron; esto se conoce como enfoque de amplitud. Otra alternativa es usar un orden aleatorio, donde el rastreador elige una URL al azar entre las que se encuentran en su cola de páginas no visitadas. Otras alternativas son usar métricas que combinan el orden con la clasificación por importancia, como el número de backlinks o el PageRank.

Analicemos cómo se puede usar PageRank para este propósito. Es necesario revisar ligeramente la fórmula de PageRank presentada anteriormente. Ahora modelamos un usuario aleatorio: al llegar a una página P , es probable que elija una de las URL de esta página como la siguiente que visite con una probabilidad (igual) d , o que acceda a una página aleatoria con una probabilidad $1-d$. Por lo tanto, la fórmula anterior para PageRank se revisa de la siguiente manera:

$$PR(P_i) = (1 - d) + d \frac{\sum_{P_j \in B_Pi} PR(P_j)}{|B_Pi|}$$

La ordenación de las URL según esta fórmula permite incorporar la importancia de una página en el orden de visita. En algunas formulaciones, el primer término se normaliza con respecto al número total de páginas de la web.

Ejemplo 12.1 Considere el grafo web de la Fig. 12.3 , donde cada página web P_i es un vértice y existe una arista dirigida de P_i a P_j si P_i tiene un enlace a P_j . Suponiendo el valor comúnmente aceptado de $d = 0,85$, el PageRank de P_2 es $PR(P_2) = 0,15 + PR(P_1)$

PR(P3)
0.85(). Esta es una fórmula recursiva que se evalúa inicialmente + 2 3 asignando a cada página valores de PageRank iguales (en este caso 0.5).

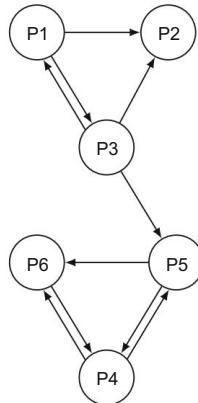


Fig. 12.3 Representación gráfica web para el cálculo de PageRank

páginas) e iterar para calcular cada PR(P_i) hasta que se alcanza un punto fijo (es decir, los valores ya no cambian).

Dado que muchas páginas web cambian con el tiempo, el rastreo es una actividad continua y es necesario revisarlas. En lugar de empezar desde cero cada vez, es preferible revisar las páginas web de forma selectiva y actualizar la información recopilada.

Los rastreadores que siguen este enfoque se denominan incrementales. Garantizan que la información de sus repositorios sea lo más actualizada posible. Los rastreadores incrementales pueden determinar las páginas que revisitán según su frecuencia de cambio o mediante un muestreo de varias páginas. Los enfoques basados en la frecuencia de cambio utilizan una estimación de la frecuencia de cambio de una página para determinar la frecuencia con la que debe revisarse. Intuitivamente, se podría asumir que las páginas con alta frecuencia de cambio deberían visitarse con mayor frecuencia, pero esto no siempre es cierto: cualquier información extraída de una página que cambia con frecuencia probablemente se vuelva obsoleta rápidamente, y puede ser mejor aumentar el intervalo de revisita de esa página. También es posible desarrollar un rastreador incremental adaptativo de modo que el rastreo de un ciclo se vea afectado por la información recopilada en el ciclo anterior. Los enfoques basados en muestreo se centran en sitios web en lugar de páginas web individuales. Se muestrea un pequeño número de páginas de un sitio web para estimar cuánto ha cambiado. Con base en esta estimación de muestreo, el rastreador determina la frecuencia con la que debe visitar ese sitio.

Algunos motores de búsqueda se especializan en buscar páginas relacionadas con un tema específico. Estos motores utilizan rastreadores optimizados para el tema objetivo, conocidos como rastreadores especializados. Un rastreador especializado clasifica las páginas según su relevancia para el tema objetivo y las utiliza para determinar qué páginas debe visitar a continuación. Las técnicas de clasificación ampliamente utilizadas en la recuperación de información se emplean para evaluar la relevancia; las técnicas de aprendizaje se emplean para identificar el tema de una página determinada. Estas técnicas quedan fuera de nuestro alcance, pero varias se han desarrollado para este fin, como el clasificador Bayesiano Naïve y sus extensiones, el aprendizaje por refuerzo, entre otras.

Para lograr un escalamiento razonable, el rastreo puede paralelizarse mediante la ejecución de rastreadores paralelos. Cualquier diseño de rastreadores paralelos debe utilizar esquemas que minimicen la sobrecarga de la paralelización. Por ejemplo, dos rastreadores ejecutándose en paralelo podrían descargar el mismo conjunto de páginas. Es evidente que esta superposición debe evitarse mediante la coordinación de las acciones de los rastreadores. Un método de coordinación utiliza un coordinador central para asignar dinámicamente a cada rastreador un conjunto de páginas para descargar. Otro esquema de coordinación consiste en particionar lógicamente la web. Cada rastreador conoce su partición, por lo que no se requiere coordinación central. Este esquema se conoce como asignación estática.

12.2.2 Indexación

Para buscar eficientemente en las páginas rastreadas y la información recopilada, se construyen varios índices, como se muestra en la Fig. 12.2. Los dos índices más importantes son el índice de estructura (o de enlaces) y el índice de texto (o de contenido).

12.2.2.1 Índice de estructura

El índice de estructura se basa en el modelo gráfico que analizamos en la sección 12.1, donde el gráfico representa la estructura de la parte rastreada de la web. El almacenamiento y la recuperación eficientes de estas páginas son importantes, y en la sección 12.1 se analizaron dos técnicas para abordar estos problemas. El índice de estructura puede utilizarse para obtener información importante sobre la vinculación de páginas web, como información sobre la vecindad de una página y sus páginas hermanas.

12.2.2.2 Índice de texto

El índice más importante y más utilizado es el índice de texto. Los índices que permiten la recuperación basada en texto pueden implementarse mediante cualquiera de los métodos de acceso tradicionales para buscar en colecciones de documentos de texto. Algunos ejemplos son las matrices de sufijos, los archivos o índices invertidos y los archivos de firmas. Aunque no se abordará en profundidad todos estos índices, analizaremos cómo se utilizan los índices invertidos en este contexto, ya que son los índices de texto más populares.

Un índice invertido es una colección de listas invertidas, donde cada lista está asociada a una palabra específica. En general, una lista invertida para una palabra dada es una lista de identificadores de documento donde aparece esa palabra. La ubicación de la palabra en una página específica también se puede guardar como parte de la lista invertida. Esta información suele ser necesaria en consultas de proximidad y en la clasificación de resultados de consultas. Los algoritmos de búsqueda también suelen utilizar información adicional sobre la ocurrencia de términos en una página web. Por ejemplo, los términos que aparecen en negrita (dentro de etiquetas fuertes), en encabezados de sección (dentro de etiquetas H1 o H2 en HTML) o como texto de anclaje podrían tener una ponderación diferente en los algoritmos de clasificación.

Además de la lista invertida, muchos índices de texto también mantienen un léxico, que es una lista de todos los términos que aparecen en el índice. El léxico también puede contener estadísticas a nivel de término que pueden ser utilizadas por algoritmos de clasificación.

La construcción y el mantenimiento de un índice invertido presentan tres dificultades principales:

1. En general, crear un índice invertido implica procesar cada página, leer todas las palabras y almacenar la ubicación de cada una. Finalmente, los archivos invertidos se escriben en el disco. Este proceso, si bien es trivial para colecciones pequeñas y estáticas, se vuelve difícil de gestionar cuando se trabaja con una colección extensa y no estática como la web.
2. La rápida evolución de la web supone un reto para mantener la frescura del índice. Si bien en la sección anterior argumentamos que se deben implementar rastreadores incrementales para garantizar la frescura, la reconstrucción periódica del índice sigue siendo necesaria, ya que la mayoría de las técnicas de actualización incremental no funcionan bien al gestionar los grandes cambios que suelen observarse entre rastreos sucesivos.
3. Los formatos de almacenamiento de índices invertidos deben diseñarse cuidadosamente. Existe un equilibrio entre la mejora del rendimiento mediante un índice comprimido que permite almacenar partes del índice en caché y la sobrecarga de la descompresión en tiempo de consulta. Lograr el equilibrio adecuado se convierte en una preocupación fundamental al trabajar con colecciones a escala web.

Para abordar estos desafíos y desarrollar un índice de texto altamente escalable, se puede distribuir el índice, ya sea creando un índice invertido local en cada máquina donde se ejecuta el motor de búsqueda o creando un índice invertido global que luego se comparta. No profundizaremos en estos temas, ya que los problemas son similares a los de la gestión de datos distribuidos y directorios que ya abordamos en capítulos anteriores.

12.2.3 Clasificación y análisis de enlaces

Un motor de búsqueda típico muestra una gran cantidad de páginas web que se espera que sean relevantes para la consulta del usuario. Sin embargo, es probable que estas páginas difieran en calidad y relevancia. No se espera que el usuario navegue por esta extensa colección para encontrar una página de alta calidad. Claramente, se necesitan algoritmos que clasifiquen estas páginas de forma que las de mayor calidad aparezcan entre los primeros resultados.

Los algoritmos basados en enlaces permiten clasificar un conjunto de páginas. Repitiendo lo que comentamos antes, la intuición es que si una página P_j contiene un enlace a la página P_i , es probable que los autores de la página P_j consideren que la página P_i es de buena calidad.

Por lo tanto, una página que tiene una gran cantidad de enlaces entrantes es probablemente de alta calidad y, por lo tanto, la cantidad de enlaces entrantes a una página puede usarse como criterio de clasificación. Esta intuición es la base de los algoritmos de ranking, pero, por supuesto, cada algoritmo la implementa de forma diferente. Ya hemos hablado del algoritmo PageRank, que se utiliza para la clasificación de resultados, además del rastreo.

Discutiremos un algoritmo alternativo llamado HITS para destacar diferentes formas de abordar el tema.

HITS también es un algoritmo basado en enlaces. Se basa en la identificación de "autoridades" y "centros de referencia". Una buena página de autoridad recibe una alta clasificación. Los centros de referencia y las autoridades tienen una relación que se refuerza mutuamente: una buena autoridad es una página enlazada por muchos centros de referencia, y un buen centro es un documento que enlaza con muchas autoridades. Por lo tanto, una página enlazada por muchos centros de referencia (una buena página de autoridad) probablemente sea de alta calidad.

Comencemos con un grafo web, $G = (V, E)$, donde V es el conjunto de páginas y E es el conjunto de enlaces entre ellas. Cada página P_i en V tiene un par de pesos no negativos (a_{Pi}, h_{Pi}) que representan los valores de autoridad y de centro de P_i , respectivamente.

Los valores de autoridad y de concentrador se actualizan de la siguiente manera. Si una página P_i está enlazada por varios concentradores válidos, a_{Pi} se incrementa para reflejar todas las páginas P_j que enlazan con ella (la notación $P_j \rightarrow P_i$ significa que la página P_j enlaza con la página P_i):

$$a_{Pi} = \frac{h_{Pj}}{\{P_j | P_j \rightarrow P_i\}}$$

$$h_{Pi} = a_{Pj} \sum_{\{P_j | P_j \rightarrow P_i\}}$$

Por lo tanto, el valor de autoridad (valor central) de la página P_i es la suma de los valores centrales (valores de autoridad) de todas las páginas con vínculos de retroceso a P_i .

12.2.4 Evaluación de la búsqueda de palabras clave

Los motores de búsqueda basados en palabras clave son las herramientas más populares para buscar información en la web. Son sencillos y permiten realizar consultas difusas que pueden no tener una respuesta exacta, pero que solo se pueden responder de forma aproximada encontrando datos similares a las palabras clave. Sin embargo, existen limitaciones obvias en cuanto a lo que se puede lograr con una simple búsqueda de palabras clave. La limitación obvia es que la búsqueda de palabras clave no es lo suficientemente potente como para expresar consultas complejas. Esto se puede solucionar (parcialmente) mediante el uso de consultas iterativas, donde las consultas previas del mismo usuario se pueden utilizar como contexto para las consultas posteriores. Una segunda limitación es que la búsqueda de palabras clave no ofrece una visión global de la información en la web, como lo hacen las consultas a bases de datos que explotan la información del esquema de la base de datos. Si bien se puede argumentar que un esquema carece de sentido para los datos web, la falta de una visión global de los datos sigue siendo un problema. Un tercer problema es la dificultad para captar la intención del usuario mediante una simple búsqueda de palabras clave; los errores en la elección de palabras clave pueden resultar en la obtención de muchas respuestas irrelevantes.

La búsqueda por categorías aborda uno de los problemas de la búsqueda por palabras clave: la falta de una visión global de la web. La búsqueda por categorías también se conoce como directorio web, catálogo, páginas amarillas y directorios temáticos. Existen varios sitios web públicos.

directorios disponibles como la Biblioteca Virtual World Wide Web (<http://vlib.org>).² El directorio web es una taxonomía jerárquica que clasifica el conocimiento humano.

Si bien la taxonomía normalmente se muestra como un trie, en realidad es un gráfico acíclico dirigido, ya que algunas categorías tienen referencias cruzadas.

Si se identifica una categoría como objetivo, entonces el directorio web es una herramienta útil.

Sin embargo, no todas las páginas web se pueden clasificar, por lo que el usuario puede usar el directorio para realizar búsquedas. Además, el procesamiento del lenguaje natural no puede ser completamente efectivo para categorizar páginas web. Necesitamos recursos humanos para evaluar las páginas enviadas, lo cual puede no ser eficiente ni escalable. Finalmente, algunas páginas cambian con el tiempo, por lo que mantener el directorio actualizado implica una sobrecarga considerable.

También se han intentado involucrar a varios motores de búsqueda al responder una consulta para mejorar la recuperación y la precisión. Un metabuscador es un servicio web que recibe una consulta del usuario y la envía a múltiples motores de búsqueda heterogéneos. El metabuscador recopila las respuestas y devuelve un resultado unificado al usuario. Tiene la capacidad de ordenar los resultados por diferentes atributos, como host, palabra clave, fecha y popularidad. Algunos ejemplos son Dogpile (<http://www.dogpile.com/>), MetaCrawler (<http://www.metacrawler.com/>), y IxQuick (<http://www.ixquick.com/>). Los distintos metabuscadores tienen diferentes formas de unificar resultados y traducir la consulta del usuario a los lenguajes de consulta específicos de cada motor de búsqueda.

El usuario puede acceder a un metabuscador a través de un software cliente o una página web. Cada motor de búsqueda cubre un porcentaje menor de la web. El objetivo de un metabuscador es cubrir más páginas web que un solo motor de búsqueda combinando diferentes motores de búsqueda.

12.3 Consultas web

Las consultas declarativas y su ejecución eficiente han sido un enfoque fundamental en la tecnología de bases de datos. Sería beneficioso que estas técnicas se pudieran aplicar a la web. De esta forma, el acceso a la web podría considerarse, en cierta medida, similar al acceso a una gran base de datos. En esta sección, analizaremos varios de los enfoques propuestos.

Existen dificultades para trasladar los conceptos tradicionales de consulta de bases de datos a los datos web. Quizás la dificultad más importante reside en que las consultas de bases de datos presuponen la existencia de un esquema estricto. Como se mencionó anteriormente, es difícil argumentar que exista un esquema para los datos web similar al de las bases de datos.³ En el mejor de los casos, los datos web son semiestructurados: los datos pueden tener cierta estructura, pero esta puede no ser tan rígida, regular o completa como la de las bases de datos, por lo que diferentes instancias de los datos pueden ser similares.

2Se proporciona una lista de estas bibliotecas en https://en.wikipedia.org/wiki/List_of_web_directories.

3Nos centramos en la web "abierta" aquí; los datos de la web profunda pueden tener un esquema, pero por lo general no lo es. accesible a los usuarios.

pero no idénticos (pueden faltar atributos, añadirse atributos o tener diferencias estructurales). Consultar datos sin esquema presenta, obviamente, dificultades inherentes.

Un segundo problema es que la web es más que datos semiestructurados (y documentos). Los enlaces que existen entre las entidades de datos web (por ejemplo, las páginas) son importantes y deben considerarse. Al igual que en la búsqueda que analizamos en la sección anterior, puede ser necesario seguir y explotar los enlaces al ejecutar consultas web. Esto requiere que los enlaces se traten como objetos de primera clase.

Una tercera dificultad importante es que no existe un lenguaje comúnmente aceptado, similar a SQL, para consultar datos web. Como mencionamos en la sección anterior, la búsqueda por palabras clave tiene un lenguaje muy simple, pero esto no es suficiente para realizar consultas más completas de datos web. Ha surgido cierto consenso sobre las construcciones básicas de dicho lenguaje (p. ej., expresiones de ruta), pero no existe un lenguaje estándar. No obstante, sí han surgido lenguajes estandarizados para modelos de datos como XML y RDF (XQuery para XML y SPARQL para RDF). Posponemos su análisis hasta la sección [12.6](#), donde nos centraremos en la integración de datos web.

12.3.1 Enfoque de datos semiestructurados

Una forma de abordar la consulta de datos web es tratarlos como una colección de datos semiestructurados. Posteriormente, se pueden utilizar los modelos y lenguajes desarrollados para este fin. Los modelos y lenguajes de datos semiestructurados no se desarrollaron originalmente para gestionar datos web; más bien, abordaron los requisitos de colecciones de datos en crecimiento que no contaban con un esquema tan estricto como sus contrapartes relacionales. Sin embargo, dado que estas características también son comunes a los datos web, estudios posteriores exploraron su aplicabilidad en este ámbito. Demostramos este enfoque utilizando un modelo específico (OEM) y un lenguaje (Lorel), pero otros enfoques, como UnQL, son similares.

OEM (Object Exchange Model) es un modelo de datos semiestructurado autodescriptivo. Autodescriptivo significa que cada objeto especifica el esquema que sigue.

Un objeto OEM se define como una cuádruple etiqueta, tipo, valor y oid. Donde etiqueta es una cadena de caracteres que describe lo que representa el objeto, tipo especifica el tipo de valor del objeto, valor es obvio y oid es el identificador del objeto que lo distingue de otros. El tipo de un objeto puede ser atómico, en cuyo caso se denomina objeto atómico, o complejo, en cuyo caso se denomina objeto complejo. Un objeto atómico contiene un valor primitivo, como un entero, un real o una cadena, mientras que un objeto complejo contiene un conjunto de otros objetos, que pueden ser atómicos o complejos. El valor de un objeto complejo es un conjunto de oid.

Ejemplo 12.2 Consideremos una base de datos bibliográfica que consta de varios documentos. En la figura [12.4](#) se muestra una instantánea de una representación OEM de dicha base de datos. Cada línea muestra un objeto OEM y la sangría se proporciona para simplificar la visualización de la estructura del objeto. Por ejemplo, la segunda línea

```

<bib, complex, {&o2, &o22, &o34}, &o1>
  <doc, complex, {&o3, &o6, &o7, &o20, &o22}, &o2>
    <authors, complex, {&o4, &o5}, &o3>
      <author, string, "M. Tamer Ozsu", &o4>
      <author, string, "Patrick Valduriez", &o5>
    <title, string, "Principles of Distributed ...", &o6>
    <chapters, complex, {&o8, &o11, &o14, &o9}, &o7>
      <chapter, complex, {&o9, &o10}, &o8>
        <heading, string, "...", &o9>
        <body, string, "...", &o10>
        ...
      <chapter, complex, {&o18, &o19}, &9>
        <heading, string, "...", &o18>
        <body, string, "...", &o19>
    <what, string, "Book", &o20>
    <price, float, 98.50, &o21>
  <doc, complex, {&o23, &o25, &o26, &o27, &o28}, &o22>
    <authors, complex, {&o24, &o4}, &o23>
      <author, string, "Yingying Tao", &o24>
    <title, string, "Mining data streams ...", &o25>
    <venue, string, "CIKM", &o26>
    <year, integer, 2009, &o27>
    <sections, complex, {&o29, &o30, &o31, &o32, &o33}, &28>
      <section, string, "...", &o29>
      ...
      <section, string, "...", &o33>
  <doc, complex, {&o16,&o9,&o7,&o18,&o19,&o20,&o21},&o34>
    <author, string, "Anthony Bonato", &o35>
    <title, string, "A Course on the Web Graph", &o36>
    <what, string, "Book", &o20>
    <ISBN, string, "TK5105.888.B667", &o37>
    <chapters, complex, {&o39, &o42, &o45}, &o38>
      <chapter, complex, {&o40, &o41}, &o39>
        <heading, string, "...", &o40>
        <body, string, "...", &o41>
      <chapter, complex, {&o43, &o44}, &o42>
        <heading, string, "...", &o43>
        <body, string, "...", &o44>
      <chapter, complex, {&o46, &o47}, &45>
        <heading, string, "...", &o46>
        <body, string, "...", &o47>
    <publisher, string, "AMS", &o48>

```

Fig. 12.4 Un ejemplo de especificación OEM

<doc, complex, &o3, &o6, &o7, &o20, &o21, &o2> define un objeto cuya etiqueta es doc, tipo es complex, oid es &o2 y cuyo valor consiste en objetos cuyos oids son &o3, &o6, &o7, &o20 y &o21.

Esta base de datos contiene tres documentos (&o2, &o22, &o34); el primero y el tercero son libros, y el segundo es un artículo. Existen puntos en común entre ambos libros (e incluso el artículo), pero también diferencias. Por ejemplo, &o2 tiene

la información de precio que 'no tiene, mientras que 'tiene información de ISBN y editor que t'o2 no tiene.

Como se mencionó anteriormente, los datos OEM son autodescriptivos, donde cada objeto se identifica a sí mismo mediante su tipo y su etiqueta. Es fácil ver que los datos OEM pueden representarse como un grafo etiquetado por vértices donde los vértices corresponden a los objetos OEM y las aristas corresponden a la relación de subobjetos. La etiqueta de un vértice es el oid y la etiqueta del vértice del objeto correspondiente. Sin embargo, es bastante común en la literatura modelar los datos como un grafo etiquetado por aristas: si el objeto o_j es un subobjeto del objeto o_i , entonces la etiqueta de o_j se asigna a la arista que conecta o_i con o_j , y los oids se omiten como etiquetas de vértice. En el Ejemplo 12.3, usamos una representación etiquetada por vértices y aristas que muestra los oids como etiquetas de vértice y asigna etiquetas de aristas como se describió anteriormente.

Ejemplo 12.3. La Figura 12.5 muestra la representación gráfica con vértices y aristas de la base de datos OEM del Ejemplo 12.2. Normalmente, cada vértice terminal (es decir, sin aristas salientes) también contiene el valor de ese objeto. Para simplificar la explicación, no se muestran los valores.

El enfoque semiestructurado es adecuado para modelar datos web, ya que puede representarse como un grafo. Además, acepta que los datos pueden tener cierta estructura, pero esta puede no ser tan rígida, regular ni completa como la de las bases de datos tradicionales. Los usuarios no necesitan conocer la estructura completa al consultar los datos. Por lo tanto, expresar una consulta no debería requerir un conocimiento completo de la estructura. Estas representaciones gráficas de los datos en cada fuente de datos se generan mediante los envoltorios que se analizaron en la sección 7.2.

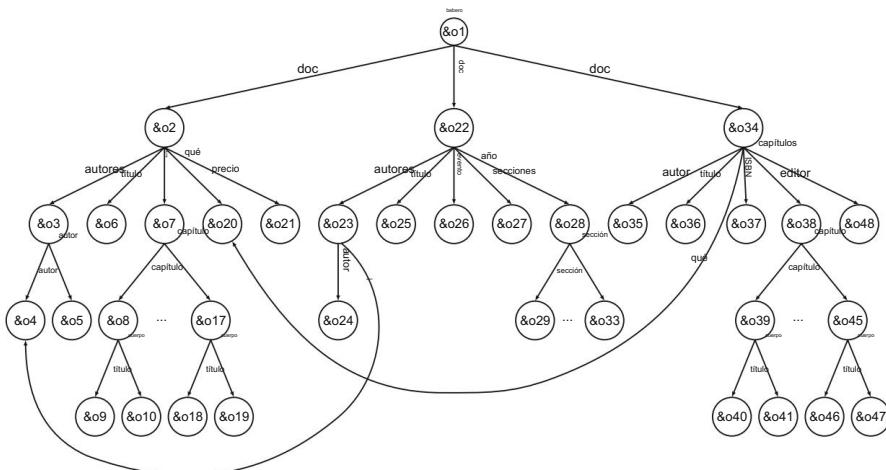


Fig. 12.5 El gráfico OEM correspondiente a la base de datos OEM del Ejemplo 12.2

Se han desarrollado varios lenguajes para consultar datos semiestructurados. Como se mencionó anteriormente, centraremos nuestra discusión en un lenguaje específico, Lorel, pero otros lenguajes presentan enfoques básicos similares.

Lorel tiene la estructura habitual SELECT-FROM-WHERE, pero permite expresiones de ruta en las cláusulas SELECT, FROM y WHERE. Por lo tanto, la construcción fundamental para la formación de consultas Lorel es una expresión de ruta. En su forma más simple, una expresión de ruta en Lorel es una secuencia de etiquetas que comienza con el nombre de un objeto o una variable que denota un objeto. Por ejemplo, bib.doc.title es una expresión de ruta cuya interpretación es comenzar en bib y seguir el doc con etiqueta de borde y luego el título con etiqueta de borde. Observe que hay tres rutas en la Fig. 12.5 que cumplirían con esta expresión: (i) &o1.doc:&o2.title:&o6, (ii) &o1.doc:&o22.title:&o25, y (iii) &o1.doc:&o34.title:&o36.

Cada una de estas se denomina ruta de datos. En Lorel, las expresiones de ruta pueden ser expresiones regulares más complejas, de modo que lo que sigue al nombre del objeto o la variable no es solo una etiqueta, sino expresiones más generales que pueden construirse mediante conjunción, disyunción (|), iteración (?) para indicar 0 o 1 ocurrencia, + para indicar 1 o más, y * para indicar 0 o más) y comodines (#).

Ejemplo 12.4 Los siguientes son ejemplos de expresiones de ruta aceptables en Lorel:

- (a) bib.doc(.authors)??.author : comienza desde bib, sigue el borde doc y el borde author con un borde author opcional en el medio. (b) bib.doc.#.author : comienza desde bib, sigue el borde doc, luego una cantidad arbitraria de bordes con etiquetas no especificadas (usando el comodín #), y sigue el borde author.
- (c) bib.doc.%price : comienza desde bib, sigue el borde doc, luego un borde cuya etiqueta tiene la cadena "price" precedida por algunos caracteres.

Ejemplo 12.5 Los siguientes son ejemplos de consultas de Lorel que utilizan algunas de las expresiones de ruta proporcionadas en el Ejemplo 12.4:

- (a) Encuentra los títulos de los documentos escritos por Patrick Valduriez.

```
SELECCIONAR D.título
DE bib.doc D
¿DÓNDE bib.doc(.authors)??.author =
"Patrick Valduriez"
```

En esta consulta, la cláusula FROM restringe el alcance a los documentos (doc) y la cláusula SELECT especifica los nodos accesibles desde los documentos siguiendo la etiqueta del título. Podríamos haber especificado el predicado WHERE como

D(.autores)??.autor = "Patrick Valduriez".

(b) Encuentra los autores de todos los libros cuyo precio sea inferior a \$100.

SELECCIONAR D(.autores)? .autor DE bib.doc D

DONDE D.qué = "Libros" Y D.precio < 100

El enfoque de datos semiestructurados para modelar y consultar datos web es simple y flexible. Además, proporciona una forma natural de gestionar la estructura de contención de los objetos web, lo que facilita, en cierta medida, la estructura de enlaces de las páginas web. Sin embargo, este enfoque también presenta deficiencias. El modelo de datos es demasiado simple: no incluye una estructura de registro (cada vértice es una entidad simple) ni admite la ordenación, ya que no existe una ordenación impuesta entre los vértices de un grafo OEM. Además, la compatibilidad con enlaces es relativamente rudimentaria, ya que el modelo o los lenguajes no diferencian entre los diferentes tipos de enlaces. Los enlaces pueden mostrar relaciones de subpartes entre objetos o conexiones entre diferentes entidades que corresponden a vértices. Estos no pueden modelarse por separado ni consultarse fácilmente.

Finalmente, la estructura del grafo puede volverse bastante compleja, lo que dificulta las consultas. Si bien Lorel ofrece diversas funciones (como comodines) para facilitar las consultas, los ejemplos anteriores indican que el usuario aún necesita conocer la estructura general de los datos semiestructurados. Los grafos OEM para bases de datos grandes pueden volverse bastante complejos, y a los usuarios les resulta difícil crear las expresiones de ruta.

La cuestión, entonces, es cómo resumir el grafo para que exista una descripción breve, similar a un esquema, que facilite las consultas. Para ello, se ha propuesto una construcción llamada DataGuide. Un DataGuide es un grafo donde cada ruta en el grafo OEM correspondiente solo ocurre una vez. Es dinámico, ya que a medida que el grafo OEM cambia, el DataGuide correspondiente se actualiza. Por lo tanto, proporciona resúmenes estructurales concisos y precisos de bases de datos semiestructuradas y puede utilizarse como un esquema ligero, útil para explorar la estructura de la base de datos, formular consultas, almacenar información estadística y optimizar las consultas.

Ejemplo 12.6 La Guía de datos correspondiente al gráfico OEM en el Ejemplo 12.3 se muestra en la Figura 12.6.

12.3.2 Enfoque del lenguaje de consulta web

Los enfoques de esta categoría abordan directamente las características de los datos web, centrándose especialmente en la gestión adecuada de los enlaces. Su punto de partida es superar las deficiencias de la búsqueda por palabras clave proporcionando abstracciones adecuadas para capturar la estructura del contenido de los documentos (como en los enfoques de datos semiestructurados).

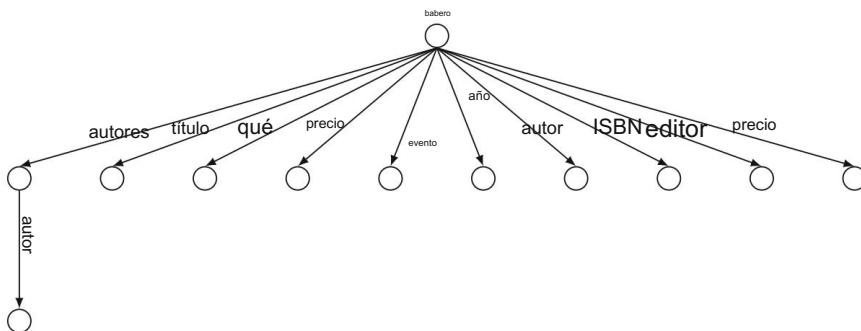


Fig. 12.6 La Guía de Datos correspondiente al gráfico OEM del Ejemplo 12.3

así como los enlaces externos. Combinan consultas basadas en contenido (p. ej., expresiones de palabras clave) y consultas basadas en estructura (p. ej., expresiones de ruta).

Se han propuesto varios lenguajes específicamente para gestionar datos web, que pueden clasificarse en lenguajes de primera y segunda generación. Los lenguajes de primera generación modelan la web como una colección interconectada de objetos atómicos. Por consiguiente, pueden expresar consultas que buscan la estructura de enlaces entre objetos web y su contenido textual, pero no pueden expresar consultas que exploten la estructura del documento de estos objetos web. Los lenguajes de segunda generación modelan la web como una colección enlazada de objetos estructurados, lo que les permite expresar consultas que explotan la estructura del documento de forma similar a los lenguajes semiestructurados. Los enfoques de primera generación incluyen WebSQL, W3QL y WebLog, mientras que los de segunda generación incluyen WebOQL y StruSQL.

Demostraremos las ideas generales considerando un lenguaje de primera generación (WebSQL) y un lenguaje de segunda generación (WebOQL).

WebSQL es uno de los primeros lenguajes de consulta que combina la búsqueda y la navegación. Aborda directamente los datos web capturados por documentos web (generalmente en formato HTML) que tienen contenido y pueden incluir enlaces a otras páginas u otros objetos (por ejemplo, archivos PDF o imágenes). Trata los enlaces como objetos de primera clase e identifica diferentes tipos de enlaces que analizaremos en breve. Como antes, la estructura puede representarse como un grafo, pero WebSQL captura la información sobre los objetos web en dos relaciones virtuales :

DOCUMENTO(URL, TÍTULO, TEXTO, TIPO, LONGITUD, MODIF)

ENLACE(BASE, HREF, ETIQUETA)

La relación DOCUMENTO contiene información sobre cada documento web. La URL identifica el objeto web y es la clave principal de la relación. TÍTULO es el título de la página web. TEXTO es el contenido textual de la página web. TIPO es el tipo de objeto web (documento HTML, imagen, etc.). LONGITUD es autoexplicativa. MODIF es la fecha de la última modificación del objeto. Excepto la URL, todos los demás atributos pueden tener valores nulos. La relación ENLACE capture la información sobre los enlaces, donde BASE es la URL.

del documento HTML que contiene el enlace, HREF es la URL del documento al que se hace referencia y LABEL es la etiqueta del enlace como se definió anteriormente.

WebSQL define un lenguaje de consulta que consta de SQL más expresiones de ruta.

Las expresiones de ruta son más poderosas que sus contrapartes en Lorel; en particular, identifican diferentes tipos de enlaces:

(a) enlace interior que existe dentro del mismo documento (#>) (b) enlace local que existe entre documentos en el mismo servidor (->) (c) enlace global que hace referencia a un documento en otro servidor (=>) (d) ruta nula (=)

Estos tipos de enlace forman el alfabeto de las expresiones de ruta. Usándolos, junto con los constructores habituales de expresiones regulares, se pueden especificar diferentes rutas, como en el Ejemplo 12.7.

Ejemplo 12.7 Los siguientes son ejemplos de posibles expresiones de ruta que se pueden especificar en WebSQL.

(a) -> | =>: una ruta de longitud uno, ya sea local o global (b) ->*: ruta local de cualquier longitud (c) =>->*: como arriba, pero en otros servidores (d) (-> |=>)*: la parte accesible de la web

Además de las expresiones de ruta que pueden aparecer en las consultas, WebSQL permite El alcance dentro de la cláusula FROM se define de la siguiente manera:

DE Relación TAL QUE condición de dominio

Donde la condición de dominio puede ser una expresión de ruta, especificar una búsqueda de texto mediante MENCIONES o especificar que un atributo (en la cláusula SELECT) es igual a un objeto web. Por supuesto, después de cada especificación de relación, podría haber una variable que se extienda a lo largo de la relación; esto es SQL estándar. Las siguientes consultas de ejemplo (con modificaciones menores) muestran las características de WebSQL.

Ejemplo 12.8 A continuación se muestran algunos ejemplos de WebSQL:

(a) El primer ejemplo que consideramos simplemente busca todos los documentos sobre "hipertexto" y demuestra el uso de MENCIONES para delimitar la consulta.

```
SELECCIONAR D.URL, D.TÍTULO
DEL DOCUMENTO D
    TAL QUE D MENCIONA "hipertexto"
    DONDE D.TYPE = "texto/html"
```

(b) El segundo ejemplo muestra dos métodos de alcance, así como una búsqueda de enlaces. La consulta busca todos los enlaces a applets de documentos sobre "Java".

SELECCIONE A.LABEL, A.HREF DEL
 DOCUMENTO D DE MODO QUE D MENCIONE "Java"
 ANCLA A TAL QUE BASE=X
 DONDE A.ETIQUETA = "applet"

- (c) El tercer ejemplo demuestra el uso de diferentes tipos de enlaces. Busca documentos que incluyan la cadena "base de datos" en su título y a los que se pueda acceder desde la página principal de la Biblioteca Digital de ACM mediante rutas de longitud dos o menos que contengan únicamente enlaces locales.

SELECCIONAR D.URL, D.TÍTULO
 DEL DOCUMENTO D TAL QUE
 "http://www.acm.org/dl"=|>|->> D DONDE D.TÍTULO CONTIENE "base de
 datos"

- (d) El último ejemplo demuestra la combinación de especificaciones de contenido y estructura en una consulta. Encuentra todos los documentos que mencionan "Informática" y todos los documentos enlazados a ellos mediante rutas de longitud dos o menos que contienen únicamente enlaces locales.

SELECCIONAR D1.URL, D1.TÍTULO, D2.URL, D2.TÍTULO
 DEL DOCUMENTO D1 TAL QUE
 D1 MENCIONES "Ciencias de la Computación",
 DOCUMENTO D2 TAL QUE D1=|>|->> D2

WebSQL puede consultar datos web basándose en los enlaces y el contenido textual de los documentos web, pero no puede consultar los documentos basándose en su estructura. Esta limitación se debe a su modelo de datos, que trata la web como una colección de objetos atómicos.

Los lenguajes de segunda generación, como WebOQL, solucionan esta deficiencia modelando la web como un grafo de objetos estructurados. En cierto modo, combinan algunas características de los enfoques de datos semiestructurados con las de los modelos de consulta web de primera generación.

La estructura de datos principal de WebOQL es un hiperárbol, un trie ordenado con aristas etiquetadas, con dos tipos de aristas: internas y externas. Una arista interna representa la estructura interna de un documento web, mientras que una arista externa representa una referencia (es decir, un hipervínculo) entre objetos. Cada arista está etiquetada con un registro que consta de varios atributos (campos). Una arista externa debe tener un atributo URL en su registro y no puede tener descendientes (es decir, son las hojas del hiperárbol).

Ejemplo 12.9. Retomemos el Ejemplo 12.2 y supongamos que, en lugar de modelar los documentos de una bibliografía, modela la colección de documentos sobre la gestión de datos en la web. Se proporciona un posible hiperárbol (parcial) para este ejemplo.

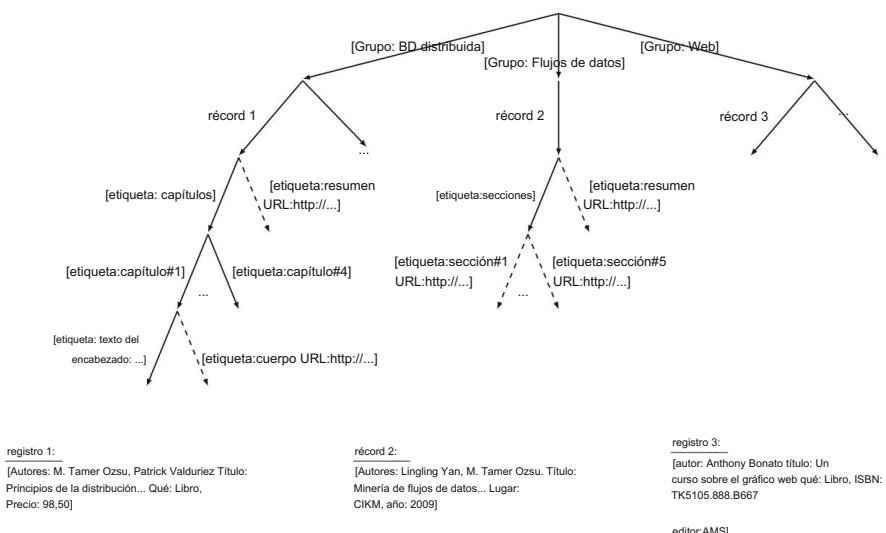


Fig. 12.7 El ejemplo del hiperárbol

En la Fig. 12.7. Nótese que hemos realizado una revisión para facilitar algunas de las consultas que se tratarán más adelante: hemos añadido un resumen a cada documento.

En la Fig. 12.7, los documentos se agrupan primero según una serie de temas, como se indica en los registros adjuntos a los bordes desde la raíz. En esta representación, los enlaces internos se muestran como bordes sólidos y los externos como bordes discontinuos.

Recuerde que en OEM (Fig. 12.5), los bordes representan tanto los atributos (p. ej., autor) como la estructura del documento (p. ej., capítulo). En el modelo WebOQL, los atributos se capturan en los registros asociados a cada borde, mientras que los bordes (internos) representan la estructura del documento.

Utilizando este modelo, WebOQL define una serie de operadores sobre árboles:

Prime: devuelve el primer subárbol de su argumento (denotado ').

Peek: extrae un campo del registro que etiqueta las primeras aristas salientes de su documento. Por ejemplo, si x apunta a la raíz del subárbol al que se llega desde la arista "Grupos = BD Distribuida", x.authors recuperaría "M. Tamer Ozsu, Patrick Valduriez".

Hang: construye un trie con etiqueta de borde con un registro formado con los argumentos (denotado como []).

Ejemplo 12.10 Supongamos que el trie mostrado en la Fig. 12.8a se recupera como resultado de una consulta (denominada Q1). La expresión ["Etiqueta: "Artículos de Ozsu" / Q1"] genera el trie mostrado en la Fig. 12.8b.

Concatenar: combina dos árboles (denotado +).

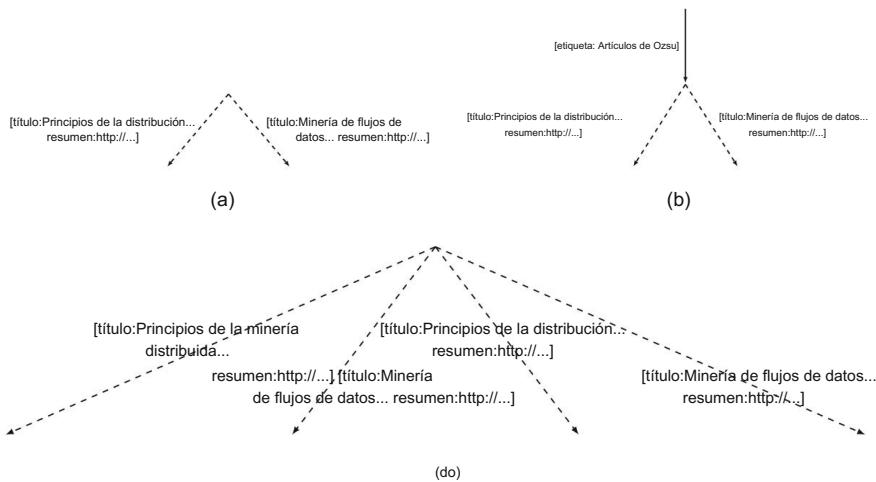


Fig. 12.8 Ejemplos de operadores Hang y Concatenate

Ejemplo 12.11 Nuevamente, asumiendo que el trie representado en la Fig. 12.8a se recupera como resultado de la consulta Q1, Q1+Q2 produce el trie de la Fig. 12.8c.

Cabeza: devuelve el primer trie simple de un trie (denotado como &). Un trie simple de un trie t son los árboles compuestos por una arista seguida de un trie (posiblemente nulo) que se origina en la raíz de t.

Cola: devuelve todos los trie excepto el primero simple de un trie (indicado con !).

Además de esto, WebOQL introduce un operador de coincidencia de patrones de cadena (denotado) cuyo argumento izquierdo es una cadena y el argumento derecho es un patrón de cadena.

Dado que el único tipo de datos admitido por el lenguaje es la cadena, este es un operador importante.

WebOQL es un lenguaje funcional, por lo que se pueden crear consultas complejas combinando estos operadores. Además, permite integrarlos en las consultas habituales de SQL (u OQL), como se muestra en el siguiente ejemplo.

Ejemplo 12.12. Sea dbDocuments los documentos de la base de datos que se muestra en la Fig. 12.7. La siguiente consulta busca los títulos y resúmenes de todos los documentos de "Ozsú", obteniendo el resultado que se muestra en la Fig. 12.8a.

```
SELECCIONAR y.ámbito, y .URL
DE x EN dbDocuments, y EN x DONDE
y.autores = "Ozsú"
```

La semántica de esta consulta es la siguiente: la variable x recorre los árboles simples de dbDocuments y, para un valor x dado, y itera sobre los árboles simples del subárbol único de x. Examina el registro de la arista y, si el valor del autor coincide con "Ozsú" (utilizando el operador de coincidencia de cadenas), construye una

trie cuya etiqueta es el atributo de título del registro al que apunta y y el valor del atributo URL del subárbol.

Los lenguajes de consulta web analizados en esta sección adoptan un modelo de datos más potente que los enfoques semiestructurados. El modelo puede capturar tanto la estructura del documento como su conectividad. Los lenguajes pueden entonces aprovechar estas diferentes semánticas de borde. Además, como hemos visto en los ejemplos de WebOQL, las consultas pueden construir nuevas estructuras como resultado. Sin embargo, la formación de estas consultas aún requiere cierto conocimiento de la estructura del grafo.

12.4 Sistemas de preguntas y respuestas

En esta sección, analizamos un enfoque interesante e inusual (desde la perspectiva de las bases de datos) para acceder a datos web: los sistemas de preguntas y respuestas (QA). Estos sistemas aceptan preguntas en lenguaje natural que se analizan para determinar la consulta específica planteada. A continuación, realizan una búsqueda para encontrar la respuesta adecuada.

Los sistemas de preguntas y respuestas han evolucionado en el contexto de los sistemas de relaciones con los usuarios (IR), cuyo objetivo es determinar la respuesta a las consultas planteadas dentro de un corpus de documentos bien definido. Estos sistemas suelen denominarse sistemas de dominio cerrado . Amplían las capacidades de las consultas de búsqueda por palabras clave de dos maneras fundamentales. En primer lugar, permiten a los usuarios especificar consultas complejas en lenguaje natural que podrían ser difíciles de especificar como simples solicitudes de búsqueda por palabras clave. En el contexto de las consultas web, también permiten formular preguntas sin un conocimiento completo de la organización de los datos. Se aplican sofisticadas técnicas de procesamiento del lenguaje natural (PLN) a estas consultas para comprender la consulta específica. En segundo lugar, buscan en el corpus de documentos y devuelven respuestas explícitas en lugar de enlaces a documentos que puedan ser relevantes para la consulta. Esto no significa que devuelvan respuestas exactas como los SGBD tradicionales, sino que pueden devolver una lista (ordenada) de respuestas explícitas a la consulta, en lugar de un conjunto de páginas web. Por ejemplo, una búsqueda de la palabra clave "Presidente de EE. UU." mediante un motor de búsqueda devolvería el resultado (parcial) de la Fig. 12.9. Se espera que el usuario encuentre la respuesta dentro de las páginas cuyas URL y descripciones cortas (denominadas fragmentos) se incluyen en esta página (y varias más). Por otro lado, una búsqueda similar utilizando la pregunta en lenguaje natural "¿Quién es el presidente de EE. UU." podría devolver una lista ordenada de nombres de presidentes (el tipo exacto de respuesta difiere entre los diferentes sistemas).

Los sistemas de preguntas y respuestas se han extendido para operar en la web. En estos sistemas, la web se utiliza como corpus (de ahí su denominación de sistemas de dominio abierto). Se accede a las fuentes de datos web mediante contenedores desarrollados específicamente para obtener respuestas a preguntas. Se han desarrollado diversos sistemas de respuesta a preguntas con diferentes objetivos y funcionalidades, como Mulder, WebQA, Start y Tritus. También existen sistemas comerciales con distintas capacidades (p. ej., Wolfram Alpha <http://www.wolframalpha.com/>).

Web Images Videos Maps News Shopping Gmail more ▾ Web History | Search settings | Sign in

Google President of USA Search Advanced Search

Web Show options Results 1 - 10 of about 138,000,000 for President of USA. (0.27 seconds)

The Presidents of the United States Short history of the US Presidency, along with biographical sketches and portraits of all the presidents to date. From the official White House site.
Abraham Lincoln - George Washington - Theodore Roosevelt
www.whitehouse.gov/about/presidents - Cached

The White House Whitehouse.gov is the official web site for the White House and President Barack Obama, the 44th President of the United States. This site is a source for ...
Contact Us - President Barack Obama - Photos & Videos
www.whitehouse.gov/ - Cached - Similar

Show more results from www.whitehouse.gov/

President of the United States, Wikipedia, the free encyclopedia For other uses, see [President of the United States](#) (disambiguation). For the list, see [List of Presidents of the United States](#).
Origin - Powers and duties - Selection process - Compensation
en.wikipedia.org/w/index.php?title=President_of_the_United_States&oldid=3500000 - Cached - Similar

List of Presidents of the United States - Wikipedia, the free ... Presidents of the United States" and "US Presidents" redirect here. For other uses, see [President of the United States](#) (disambiguation).
en.wikipedia.org/w/index.php?title=List_of_Presidents_of_the_United_States&oldid=3500000 - Cached - Similar

Show more results from en.wikipedia.org/

Image results for **President of USA** - Report images

The Presidents of the USA - EnchantedLearning.com The Presidents of the United States of America, from George Washington to George Bush.
www.enchantedlearning.com/history/us/pres/list.shtml - Cached - Similar

Presidents of the United States 7 Nov 2009 ... Presidents of the United States facts, presidential trivia, biographies, museums , scandals, speeches, research and information sources about ...
www.presidentsusa.net/ - Cached - Similar

The Presidents of the United States of America Official site with tour dates, biography, news, store and pictures.
www.presidentsrock.com/ - Similar

Presidents of the United States (POTUS) Background information, election results, cabinet members, notable events, and some points of interest on each of the presidents.
www.ipl.org/div/potus/ - Cached - Similar

Executive Office of the President USA.gov USA.gov Executive Office of the President - Links to various departments in the Executive Office of the President
www.usa.gov/agencies/federal-executive-branch - Cached - Similar

Fig. 12.9 Ejemplo de búsqueda de palabras clave

Describimos la funcionalidad general de estos sistemas utilizando la arquitectura de referencia que se muestra en la Fig. 12.10. El preprocessamiento, que no se emplea en todos los sistemas, es un proceso fuera de línea para extraer y mejorar las reglas que utilizan los sistemas. En muchos casos, se trata de análisis de documentos extraídos de la web o devueltos como respuestas a preguntas previas para determinar las estructuras de consulta más efectivas en las que se puede transformar una pregunta del usuario. Estas reglas de transformación se almacenan para su uso en tiempo de ejecución al responder a las preguntas del usuario. Por ejemplo, Tritus emplea un enfoque basado en el aprendizaje q

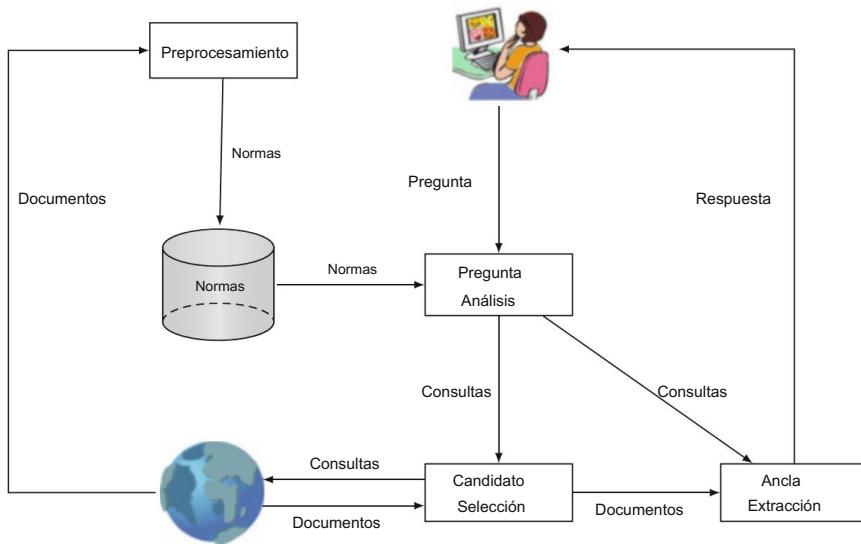


Fig. 12.10 Arquitectura general de los sistemas de control de calidad

Una colección de preguntas frecuentes y sus respuestas correctas como conjunto de datos de entrenamiento. En un proceso de tres etapas, intenta adivinar la estructura de la respuesta mediante el análisis de la pregunta y la búsqueda de la respuesta en la colección. En la primera etapa, se analiza la pregunta para extraer la frase interrogativa (p. ej., en la pregunta "¿Qué es un disco duro?", "¿Qué es un?" es una frase interrogativa). Esto se utiliza para clasificar la pregunta. En la segunda fase, analiza los pares pregunta-respuesta en los datos de entrenamiento y genera transformaciones candidatas para cada frase interrogativa (p. ej., para la frase interrogativa "¿Qué es un?", genera "se refiere a", "representa", etc.).

En la tercera etapa, cada transformación candidata se aplica a las preguntas en el conjunto de datos de entrenamiento y las consultas transformadas resultantes se envían a diferentes motores de búsqueda. Se calculan las similitudes entre las respuestas obtenidas y las respuestas reales en los datos de entrenamiento y, con base en ellas, se clasifica a las transformaciones candidatas. Las reglas de transformación clasificadas se almacenan para su uso posterior durante la ejecución de las preguntas.

La pregunta en lenguaje natural planteada por un usuario pasa primero por el proceso de análisis de preguntas. El objetivo es comprender la pregunta formulada por el usuario. La mayoría de los sistemas intentan adivinar el tipo de respuesta para categorizar la pregunta, lo cual se utiliza para traducirla en consultas y también para la extracción de respuestas. Si se ha realizado preprocesamiento, las reglas de transformación generadas se utilizan para facilitar el proceso. Si bien los objetivos generales son los mismos, los enfoques utilizados por los diferentes sistemas varían considerablemente según la sofisticación de las técnicas de PLN empleadas (esta fase suele estar relacionada exclusivamente con el PLN). Por ejemplo, el análisis de preguntas en Mulder consta de tres fases: análisis sintáctico de la pregunta, clasificación de la pregunta y generación de la consulta. Consulta

El análisis genera un trie de análisis que se utiliza para generar consultas y extraer respuestas. La clasificación de preguntas, como su nombre indica, categoriza la pregunta en una de varias clases: por ejemplo, nominal para sustantivos, numérica para números y temporal para fechas. Este tipo de categorización se utiliza en la mayoría de los sistemas de control de calidad porque facilita la extracción de respuestas. Finalmente, la fase de generación de consultas utiliza el trie de análisis generado previamente para construir una o más consultas que se pueden ejecutar para obtener las respuestas a la pregunta. Mulder utiliza cuatro métodos diferentes en esta fase.

- Conversión de verbos: El verbo auxiliar y principal se reemplaza por el verbo conjugado (p. ej., "¿Cuándo visitó Nixon China?" se convierte en "Nixon visitó China").
- Expansión de la consulta: El adjetivo en la frase interrogativa se reemplaza por su sustantivo atributo (p. ej., "¿Cuál es la altura del Monte Everest?" se convierte en "La altura del Everest es").
- Formación de frases nominales: Algunas frases nominales se citan para darles...

Juntos al motor de búsqueda en la siguiente etapa.

- Transformación: La estructura de la pregunta se transforma en la estructura del tipo de respuesta esperado ("¿Quién fue el primer estadounidense en el espacio?" se convierte en "El primer estadounidense en el espacio fue").

Mulder es un ejemplo de un sistema que utiliza un sofisticado enfoque de PLN para el análisis de preguntas. En el otro extremo del espectro se encuentra WebQA, que adopta un enfoque ligero para el análisis de preguntas.

Una vez analizada la pregunta y generadas una o más consultas, el siguiente paso es generar las respuestas candidatas. Las consultas generadas en la etapa de análisis de preguntas se utilizan en este paso para realizar una búsqueda por palabras clave de documentos relevantes. Muchos sistemas simplemente utilizan los motores de búsqueda generales en este paso, mientras que otros también consideran fuentes de datos adicionales disponibles en la web.

Por ejemplo, el World Factbook de la CIA (<https://www.cia.gov/library/publications/the-world-factbook/>) Es una fuente muy popular de datos factuales confiables sobre los países.

De manera similar, se puede obtener información meteorológica de manera muy confiable a partir de varias fuentes de datos meteorológicos, como Weather Network (<http://www.theweathernetwork.com/>), o Weather Underground (<http://www.wunderground.com/>). Estas fuentes de datos adicionales pueden proporcionar mejores respuestas en algunos casos, y los distintos sistemas las aprovechan en distintos grados. Dado que las distintas consultas se pueden responder mejor con distintas fuentes de datos (y, a veces, incluso con distintos motores de búsqueda), un aspecto importante de esta etapa de procesamiento es la elección del motor o motores de búsqueda o fuente de datos adecuados para cada consulta. La alternativa simple de enviar las consultas a todos los motores de búsqueda y fuentes de datos no es una decisión inteligente, ya que estas operaciones son bastante costosas en la web. Normalmente, la información de la categoría se utiliza para facilitar la elección de las fuentes adecuadas, junto con una lista ordenada de fuentes y motores para las diferentes categorías. Para cada motor de búsqueda y fuente de datos, es necesario crear contenedores para convertir la consulta al formato de esa fuente de datos o motor de búsqueda y convertir los documentos de resultados a un formato común para su posterior análisis.

En respuesta a las consultas, los motores de búsqueda devuelven enlaces a los documentos junto con fragmentos cortos, mientras que otras fuentes de datos devuelven resultados en una variedad de formatos.

Los resultados devueltos se normalizan en "registros". Las respuestas directas deben extraerse de estos registros, que es la función de la fase de extracción de respuestas. Se pueden utilizar varias técnicas de procesamiento de texto para hacer coincidir las palabras clave con (posiblemente partes de) los registros devueltos. Posteriormente, estos resultados deben clasificarse utilizando varias técnicas de recuperación de información (p. ej., frecuencias de palabras, frecuencia inversa de documentos). En este proceso, se utiliza la información de categoría que se genera durante el análisis de la pregunta. Diferentes sistemas emplean diferentes nociones de la respuesta apropiada. Algunos devuelven una lista ordenada de respuestas directas (p. ej., si la pregunta es "¿Quién inventó el teléfono?", devolverían "Alexander Graham Bell" o "Graham Bell" o "Bell", o todas ellas en orden de clasificación⁴), mientras que otros devuelven un orden de clasificación de la parte de los registros que contienen las palabras clave en la consulta (es decir, un resumen de la parte relevante del documento).

Los sistemas de preguntas y respuestas son muy diferentes a otros enfoques de consulta web que hemos analizado en secciones anteriores. Ofrecen mayor flexibilidad a los usuarios para realizar consultas sin necesidad de conocer la organización de los datos web. Por otro lado, se ven limitados por las peculiaridades del lenguaje natural y las dificultades de su procesamiento.

12.5 Búsqueda y consulta en la Web oculta

Actualmente, la mayoría de los motores de búsqueda de propósito general solo operan en la web oculta (PIW), mientras que una cantidad considerable de datos valiosos se almacena en bases de datos ocultas, ya sea como datos relacionales, documentos incrustados o en muchos otros formatos. La tendencia actual en las búsquedas web es encontrar maneras de explorar tanto la web oculta como la PIW, por dos razones principales. En primer lugar, el tamaño: la web oculta (en términos de páginas HTML generadas) es considerablemente mayor que la PIW; por lo tanto, la probabilidad de encontrar respuestas a las consultas de los usuarios es mucho mayor si también se puede buscar en la web oculta. En segundo lugar, la calidad de los datos: los datos almacenados en la web oculta suelen ser de mucha mayor calidad que los que se encuentran en las páginas web públicas, ya que están correctamente seleccionados. Si se puede acceder a ellos, se puede mejorar la calidad de las respuestas.

Sin embargo, la búsqueda en la web oculta enfrenta muchos desafíos, el más importante de los cuales es... de los cuales son los siguientes:

1. Los rastreadores comunes no se pueden utilizar para buscar en la web oculta, ya que existen Ni páginas HTML, ni hipervínculos para rastrear.
2. Por lo general, solo se puede acceder a los datos de las bases de datos ocultas mediante una búsqueda. interfaz o una interfaz especial, que requiere acceso a esta interfaz.
3. En la mayoría de los casos (si no en todos), se desconoce la estructura subyacente de la base de datos y los proveedores de datos suelen ser reacios a proporcionar cualquier información sobre sus datos que pueda ayudar en el proceso de búsqueda (posiblemente debido a la sobrecarga).

⁴El inventor del teléfono es objeto de controversia, con múltiples reivindicaciones de la invención. En este ejemplo utilizaremos a Bell, ya que fue el primero en patentar el dispositivo.

de recopilar esta información y mantenerla). Es necesario trabajar a través de las interfaces proporcionadas por estas fuentes de datos.

En el resto de esta sección, analizamos estas cuestiones así como algunas soluciones propuestas.

12.5.1 Explorando la Web oculta

Una forma de abordar el problema de la búsqueda en la web oculta es intentar rastrear de forma similar a la del PIW. Como ya se mencionó, la única manera de acceder a las bases de datos web ocultas es a través de sus interfaces de búsqueda. Un rastreador web oculto debe ser capaz de realizar dos tareas: (a) enviar consultas a la interfaz de búsqueda de la base de datos y (b) analizar las páginas de resultados y extraer información relevante.

12.5.1.1 Consulta de la interfaz de búsqueda

Un enfoque consiste en analizar la interfaz de búsqueda de la base de datos y crear una representación interna. Esta representación especifica los campos utilizados en la interfaz, sus tipos (p. ej., cuadros de texto, listas, casillas de verificación, etc.), sus dominios (p. ej., valores específicos, como en las listas, o simplemente cadenas de texto libre, como en los cuadros de texto) y las etiquetas asociadas a estos campos. Extraer estas etiquetas requiere un análisis exhaustivo de la estructura HTML de la página.

A continuación, esta representación se compara con la base de datos específica de la tarea del sistema. La coincidencia se basa en las etiquetas de los campos. Cuando una etiqueta coincide, el campo se rellena con los valores disponibles. El proceso se repite para todos los valores posibles de todos los campos del formulario de búsqueda, y el formulario se envía con cada combinación de valores y se recuperan los resultados.

Otro enfoque consiste en utilizar tecnología de agentes. En este caso, se desarrollan agentes web ocultos que interactúan con los formularios de búsqueda y recuperan las páginas de resultados. Esto implica tres pasos: (a) encontrar los formularios, (b) aprender a rellenarlos y (c) identificar y recuperar las páginas de resultados.

El primer paso se logra partiendo de una URL (un punto de entrada), recorriendo enlaces y utilizando heurísticas para identificar páginas HTML que contienen formularios, excluyendo aquellas que contienen campos de contraseña (p. ej., páginas de inicio de sesión, registro o compra). La tarea de llenar el formulario depende de la identificación de etiquetas y su asociación con los campos del formulario. Esto se logra mediante heurísticas sobre la ubicación de la etiqueta en relación con el campo (a la izquierda o encima de él). Con las etiquetas identificadas, el agente determina el dominio de aplicación al que pertenece el formulario y rellena los campos con valores de ese dominio según las etiquetas (los valores se almacenan en un repositorio accesible para el agente).

12.5.1.2 Análisis de las páginas de resultados

Una vez enviado el formulario, la página devuelta debe analizarse, por ejemplo, para determinar si es una página de datos o una página de refinamiento de búsqueda. Esto se puede lograr comparando los valores de esta página con los del repositorio del agente. Una vez encontrada una página de datos, se recorre, junto con todas las páginas que enlaza (especialmente las que tienen más resultados), hasta que no se encuentren más páginas que pertenezcan al mismo dominio.

Sin embargo, las páginas devueltas suelen contener muchos datos irrelevantes, además de los resultados reales, ya que la mayoría de las páginas de resultados siguen una plantilla con una cantidad considerable de texto utilizado únicamente con fines de presentación. Un método para identificar plantillas de páginas web consiste en analizar el contenido textual y las estructuras de etiquetas adyacentes de un documento para extraer datos relacionados con la consulta. Una página web se representa como una secuencia de segmentos de texto, donde un segmento de texto es una etiqueta encapsulada entre dos etiquetas. El mecanismo para detectar plantillas es el siguiente:

1. Los segmentos de texto de los documentos se analizan en función del contenido textual y los segmentos de etiquetas adyacentes.
2. Se identifica una plantilla inicial examinando los dos primeros documentos de muestra.
3. Luego se genera la plantilla si se combinan los segmentos de texto con sus adyacentes.
Los segmentos de etiquetas se encuentran en ambos documentos.
4. Los documentos recuperados posteriormente se comparan con la plantilla generada. Los segmentos de texto que no se encuentran en la plantilla se extraen de cada documento para su posterior procesamiento.
5. Cuando no se encuentran coincidencias en la plantilla existente, se elimina el contenido del documento, extraído para la generación de futuras plantillas.

12.5.2 Metabúsqueda

La metabúsqueda es otro enfoque para consultar la web oculta. Dada la consulta de un usuario, un metabuscador realiza las siguientes tareas:

1. Selección de bases de datos: seleccionar las bases de datos más relevantes para la consulta del usuario. Esto requiere recopilar información sobre cada base de datos. Esta información se conoce como resumen de contenido, que consiste en información estadística y suele incluir la frecuencia de las palabras que aparecen en la base de datos.
2. Traducción de consultas: traducir la consulta a un formato adecuado para cada base de datos (por ejemplo, llenando determinados campos en la interfaz de búsqueda de la base de datos).
3. Fusión de resultados: recopilar los resultados de las distintas bases de datos, fusionarlos (y muy probablemente, ordenarlos) y devolvérselos al usuario.

A continuación analizamos con más detalle las fases importantes de la metabúsqueda.

12.5.2.1 Extracción del resumen de contenido

El primer paso en la metabúsqueda es calcular los resúmenes de contenido. En la mayoría de los casos, los proveedores de datos no están dispuestos a proporcionar esta información. Por lo tanto, el propio metabuscador la extrae.

Un enfoque posible es extraer un conjunto de muestras de documentos de una base de datos dada D y calcular la frecuencia de cada palabra observada w en la muestra, SampleDF (w).

La técnica funciona de la siguiente manera:

1. Comience con un resumen de contenido vacío donde SampleDF (w) = 0 para cada palabra w, y un diccionario de palabras general (es decir, no específico de D) y completo.
2. Elija una palabra y envíela como consulta a la base de datos D.
3. Recupere los k primeros documentos de entre los documentos devueltos.
4. Si el número de documentos recuperados excede un umbral preestablecido, deténgase.

De lo contrario, continúe el proceso de muestreo volviendo al Paso 2.

Existen dos versiones principales de este algoritmo que difieren en la ejecución del paso 2. Uno de ellos selecciona una palabra aleatoria del diccionario. El segundo algoritmo selecciona la siguiente consulta entre las palabras ya descubiertas durante el muestreo. El primero construye mejores perfiles, pero es más costoso.

Una alternativa es utilizar una técnica de sondeo focalizado que clasifique las bases de datos en una categorización jerárquica. La idea es preclasificar un conjunto de documentos de entrenamiento en categorías y, a continuación, extraer diferentes términos de estos documentos y utilizarlos como sondas de consulta para la base de datos. Las sondas de una sola palabra se utilizan para determinar las frecuencias reales de estas palabras en los documentos, mientras que solo se calculan las frecuencias de los documentos de muestra para otras palabras que aparecen en sondas más largas. Estas se utilizan para estimar las frecuencias reales de estas palabras en los documentos.

Otro enfoque consiste en comenzar seleccionando aleatoriamente un término de la propia interfaz de búsqueda, asumiendo que, con gran probabilidad, este término estará relacionado con el contenido de la base de datos. Se consulta la base de datos buscando este término y se recuperan los k documentos principales. A continuación, se selecciona aleatoriamente un término posterior entre los términos extraídos de los documentos recuperados. El proceso se repite hasta obtener un número predefinido de documentos, y luego se calculan las estadísticas basadas en ellos.

12.5.2.2 Categorización de bases de datos

Un buen enfoque que puede facilitar el proceso de selección de bases de datos es categorizarlas en varias categorías (por ejemplo, como directorio de Yahoo). La categorización facilita la localización de una base de datos dada la consulta de un usuario y hace que la mayoría de los resultados devueltos sean relevantes para la consulta.

Si se utiliza la técnica de sondeo enfocado para generar resúmenes de contenido, entonces el mismo algoritmo puede sondear cada base de datos con consultas de alguna categoría y

Contar el número de coincidencias. Si el número de coincidencias supera un umbral determinado, se dice que la base de datos pertenece a esta categoría.

Selección de base de datos

La selección de bases de datos es crucial en el proceso de metabúsqueda, ya que tiene un impacto crucial en la eficiencia y eficacia del procesamiento de consultas en múltiples bases de datos. Un algoritmo de selección de bases de datos intenta encontrar el mejor conjunto de bases de datos, basándose en la información sobre su contenido, para ejecutar una consulta determinada. Generalmente, esta información incluye el número de documentos que contienen cada palabra (conocido como frecuencia de documento), así como otras estadísticas sencillas relacionadas, como el número de documentos almacenados en la base de datos.

Dados estos resúmenes, un algoritmo de selección de bases de datos estima la relevancia de cada base de datos para una consulta determinada (por ejemplo, en términos de la cantidad de coincidencias que se espera que cada base de datos produzca para la consulta).

GIOSS es un algoritmo sencillo de selección de bases de datos que asume que las palabras de consulta se distribuyen de forma independiente entre los documentos de la base de datos para estimar el número de documentos que coinciden con una consulta dada. GIOSS es un ejemplo de una amplia familia de algoritmos de selección de bases de datos que se basan en resúmenes de contenido. Además, los algoritmos de selección de bases de datos esperan que dichos resúmenes de contenido sean precisos y estén actualizados.

El algoritmo de sondeo enfocado, descrito anteriormente, aprovecha la categorización de la base de datos y los resúmenes de contenido para la selección de bases de datos. Este algoritmo consta de dos pasos básicos: (1) propagar los resúmenes de contenido de la base de datos a las categorías del esquema de clasificación jerárquica, y (2) utilizar los resúmenes de contenido de las categorías y bases de datos para realizar la selección jerárquica de bases de datos, centrándose en las partes más relevantes de la jerarquía temática. Esto genera respuestas más relevantes a la consulta del usuario, ya que solo provienen de bases de datos que pertenecen a la misma categoría que la consulta.

Una vez seleccionadas las bases de datos relevantes, se consulta cada una de ellas y Los resultados devueltos se fusionan y se envían de vuelta al usuario.

12.6 Integración de datos web

En el capítulo 7, analizamos la integración de bases de datos, cada una con esquemas bien definidos. Las técnicas descritas en ese capítulo son generalmente apropiadas cuando se trata de datos empresariales. Cuando deseamos proporcionar acceso integrado a fuentes de datos web, el problema se vuelve más complejo: todas las características del "big data" influyen. En particular, los datos pueden no tener un esquema adecuado, y si lo tienen, las fuentes de datos son tan variadas que los esquemas son muy diferentes, lo que dificulta la coincidencia de esquemas. Además, la cantidad de datos, e incluso el número de fuentes de datos, es significativamente mayor que en un entorno empresarial, lo que hace que la curación manual sea prácticamente imposible. La calidad de los datos en la web también es...

mucho más sospechosos que las recopilaciones de datos empresariales que consideramos anteriormente, y esto aumenta la importancia de las soluciones de limpieza de datos.

Un enfoque adecuado para la integración de datos web es la denominada integración de pago por uso, donde la inversión inicial para integrar los datos se reduce significativamente, eliminando algunas de las etapas descritas en el capítulo 7. En su lugar, se proporciona un marco y una infraestructura básica para que los propietarios de datos integren fácilmente sus conjuntos de datos en una federación. Una propuesta para el enfoque de pago por uso para la integración de datos web son los espacios de datos, que abogan por una plataforma de integración ligera con opciones de acceso quizás rudimentarias (p. ej., búsqueda por palabras clave) para comenzar, y formas de mejorar el valor de la integración con el tiempo, brindando la oportunidad de desarrollar herramientas para un uso más sofisticado. Quizás los lagos de datos que han comenzado a recibir atención y que analizamos en el capítulo 10, sean versiones más avanzadas de la propuesta de espacios de datos.

En esta sección, abordamos algunos de los enfoques desarrollados para abordar estos desafíos. En particular, analizaremos las tablas web y las tablas de fusión (Sección 12.6.1) como un enfoque de integración de bajo consumo para datos estructurados tabulares. Posteriormente, analizaremos la web semántica y el enfoque de Datos Abiertos Vinculados (LOD) para la integración de datos web (Sección 12.6.2.3). Finalmente, en la Sección 12.6.3, abordaremos los problemas de limpieza de datos y el uso de técnicas de aprendizaje automático en la integración y limpieza de datos a escala web.

12.6.1 Tablas web/Tablas de fusión

Dos enfoques populares para la integración ligera de datos web son los portales web y los mashups , que agregan datos web y de otro tipo sobre temas específicos, como viajes, reservas de hotel, etc. Ambos difieren en las tecnologías que utilizan, pero esto no es relevante para nuestro análisis. Estos son ejemplos de sistemas de "integración vertical" donde cada mashup o portal se dirige a un dominio.

Una primera pregunta que surge al desarrollar un mashup es cómo encontrar los datos web relevantes. El proyecto de tablas web es un intento temprano de encontrar datos en la web con estructura de tablas relacionales y proporcionar acceso a través de ellas (las llamadas tablas "similares a bases de datos"). El enfoque se centra en la web abierta, y las tablas en la web profunda no se consideran, ya que su descubrimiento es un problema mucho más complejo. Incluso encontrar tablas similares a bases de datos en la web abierta no es fácil, ya que las estructuras de tablas relacionales habituales (es decir, los nombres de los atributos) pueden no existir. Las tablas web emplean un clasificador que puede agrupar las tablas HTML en relacionales y no relacionales. A continuación, proporciona herramientas para extraer un esquema y mantener estadísticas sobre los esquemas que se pueden utilizar en la búsqueda en estas tablas. Se introducen oportunidades de unión entre tablas para permitir una navegación más sofisticada a través de las tablas descubiertas. Las tablas web pueden considerarse un método para recuperar y consultar datos web, pero también sirven como un marco de integración virtual para datos web con información de esquema global.

El proyecto Fusion Tables de Google lleva las tablas web un paso más allá al permitir que los usuarios carguen sus propias tablas (en varios formatos) además de las descubiertas.