Unidad 1

Bases de datos

Una Base de Datos es una colección de datos interrelacionados. Es una colección de archivos diseñados para servir a múltiples aplicaciones.

Sistema Manejador de Base de Datos (DBMS)

Un DBMS es una colección de datos y programas para acceder a ellos. Es una colección de programas que permiten crear y mantener una base de datos.

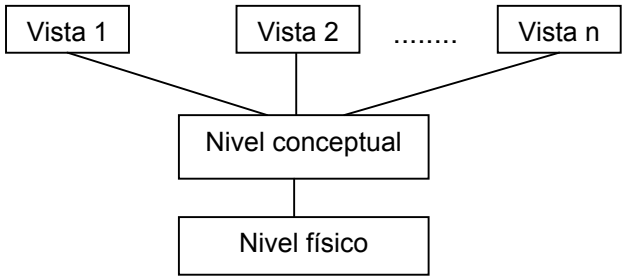
Los objetivos de un DBMS son:

- El objetivo principal de un DBMS es proporcionar un entorno que sea a la vez conveniente y eficiente para ser utilizado al extraer y almacenar información de la base de datos.
- Evitar redundancia e inconsistencia de datos: la redundancia aumenta los costos de almacenamiento y acceso. Puede llevar a la inconsistencia de los datos.
- Permitir acceso a los datos en todo momento: deben desarrollarse sistemas de recuperación de datos para uso general.
- Evitar anomalías en el acceso concurrente: prevenir inconsistencias al permitir que varios usuarios actualicen los datos simultáneamente.
- Evitar problemas de seguridad: no todos los usuarios del sistema de base de datos deben poder acceder a todos los datos.
- Evitar problemas de integridad en los datos: los valores de datos almacenados en la base de datos deben satisfacer ciertos tipos de restricciones de consistencia.

Abstracción de datos

Un objetivo importante de un sistema de bases de datos es proporcionar a los usuarios una visión abstracta de los datos, ocultando complejidad a través de diversos niveles de abstracción.

- Nivel físico: es el nivel más bajo, describe en detalle como se almacenan realmente los datos.
- Nivel conceptual: describe que datos son realmente almacenados en la base de datos y las relaciones que existen entre los datos. Este nivel de abstracción lo usan los administradores de bases de datos, quienes deben decidir que información se va a guardar en la base de datos.
- Nivel de visión: es el nivel más alto de abstracción, describe solo parte de la base de datos completa según los requerimientos de cada usuario. El sistema puede proporcionar muchas visiones para la misma base de datos.



Modelos de datos

Colección de herramientas conceptuales para describir datos, relaciones entre ellos, semántica asociada a los datos y restricciones de consistencia. Se dividen en tres grupos: modelos lógicos basados en objetos, modelos lógicos basados en registros y modelos físicos de datos.

Modelos lógicos basados en objetos

Se usan para describir datos en los niveles conceptual y de visión. Proporcionan capacidad de estructuración bastante flexible y permiten especificar restricciones de datos explícitamente. Algunos de los más conocidos son: modelo entidad-relación, modelo orientado a objetos, modelo binario, etc.

Modelos lógicos basados en registros

Se utilizan para describir datos en los niveles conceptual y físico. A diferencia del anterior se usan para especificar la estructura lógica global de la base de datos y para proporcionar una descripción a nivel más alto de la implementación.

Los modelos basados en registros se llaman así porque la base de datos está estructurada en registros de formato fijo de varios tipos.

Los tres modelos más aceptados son:

- Modelo relacional: representa los datos y las relaciones entre los datos mediante una colección de tablas, cada una de las cuales tiene un número de columnas con nombres únicos.
- Modelo de red: los datos se representan mediante colecciones de registros y las relaciones entre los datos se representan mediante enlaces, los cuales pueden verse como punteros. Los registros en la base de datos se organizan como colecciones de grafos arbitrarios.
- Modelo jerárquico: es similar al modelo de red. Se diferencian en que los registros están organizados como colecciones de árboles en vez de grafos arbitrarios.

Modelos físicos de datos

Se usan para describir datos en el nivel más bajo. Los dos más conocidos son el modelo unificador y memoria de elementos.

Independencia de datos

Es la capacidad de modificar la definición de un esquema en un nivel sin afectar la definición de un esquema en el nivel superior siguiente.

- Independencia física: es la capacidad de modificar el esquema físico sin provocar que se vuelvan a escribir los programas de aplicación.
- Independencia lógica: es la capacidad de modificar el esquema conceptual sin provocar que se vuelvan a escribir los programas de aplicación.

La independencia lógica es más difícil de lograr que la independencia física, ya que los programas de aplicación son fuertemente dependientes de la estructura lógica de los datos a los que acceden.

Categorías de soft de procesamiento de datos

- Sin independencia de datos: los programas de aplicación actúan directamente sobre el disco rígido, sin el sistema operativo.
- Independencia física: los programas interactúan contra un sistema operativo y este es el encargado de interactuar con el disco rígido.
- Independencia lógica parcial: se puede traer un registro y se puede leer el siguiente.
- Independencia lógica y física: se puede leer el siguiente, mediante una DBMS, siendo ese registro el que se desea. El DBMS interactúa con el sistema operativo.
- Independencia geográfica: se puede recuperar la información, sin importar donde está (físicamente). Bases de datos distribuidas.

Componentes de un DBMS

DDL (Data Definition Languaje):

Un esquema de base de datos se especifica por medio de un conjunto de definiciones que se expresan mediante un lenguaje especial llamado lenguaje de definición de datos. El resultado de la compilación de sentencias de DDL, es un conjunto de tablas las cuales se almacenan en un archivo especial llamado diccionario de datos. Este archivo contiene metadatos, datos sobre datos.

DML (Data Manipulation Languaje)

Lenguaje de manipulación de los datos. Es un lenguaje que capacita a los usuarios a acceder o manipular datos según estén organizados por el modelo de datos adecuado. Existen básicamente dos tipos:

- Procedimentales: el usuario especifica qué datos se necesitan y cómo obtenerlos.
- No procedimentales: el usuario especifica qué datos se necesitan y no cómo obtenerlos.

Los DML no procedimentales normalmente son más sencillos de aprender y usar que los procedimentales. Sin embargo, puesto que el usuario no tiene que especificar cómo conseguir los datos, estos lenguajes pueden generar código que no sea tan eficiente como el producido por los lenguajes procedimentales.

Una consulta es una sentencia que solicita la recuperación de información. El trozo de un DML que implica recuperación de información se llama lenguaje de consulta.

Gestor de Base de Datos

Es un módulo de programa que proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y consultas. Es responsable de las siguientes tareas:

- Interactuar con el gestor de archivos: el gestor de base de datos traduce distintas sentencias DML a comandos del sistema de archivos de bajo nivel.
- Implantación de la integridad: los valores de los datos que se almacenan en la base de datos deben satisfacer ciertos tipos de restricciones de consistencia. El administrador de la base de datos debe especificar explícitamente estas restricciones. El gestor de la base de datos entonces puede determinar si las actualizaciones a la base de datos dan como resultado la violación de la restricción; si es así, se debe tomar la acción apropiada.
- Implantación de la seguridad: no todos los usuarios de la base de datos necesitan tener acceso a todo su contenido.
- Copias de seguridad y recuperación: El gestor de la base de datos debe detectar fallos y restaurar la base de datos al estado que existía antes de ocurrir el fallo.
- Control de concurrencia: se debe controlar la interacción entre los usuarios concurrentes.

Administrador de Base de Datos

Es la persona que tiene el control central sobre los datos y de los programas que acceden a los datos del sistema. Sus funciones son:

- Definir el esquema: el esquema original de la base de datos se crea escribiendo un conjunto de definiciones que son traducidas por el compilador de DDL a un conjunto de tablas que son almacenadas permanentemente en el diccionario de datos.
- Definir la estructura de almacenamiento y método de acceso.
- Modificar el esquema y la organización física.
- Autorizar el acceso a los datos.
- Especificar las restricciones de integridad.

Usuarios de Bases de Datos

Existen cuatro tipos distintos de usuarios diferenciados por la forma en que esperan interaccionar con el sistema:

- Programadores de aplicaciones: interaccionan con el sistema por medio de llamadas en DML, las cuales están incorporadas en un programa escrito en un lenguaje principal.

- Usuarios sofisticados: interaccionan con el sistema sin escribir programas. Escriben sus preguntas en un lenguaje de consultas de bases de datos. Cada consulta se somete a un procesador de consultas cuya función es tomar una sentencia en DML y descomponerla en instrucciones que entienda el gestor de la base de datos.
- Usuarios especializados: estos usuarios escriben aplicaciones de bases de datos especializadas que no encajan en el marco tradicional de procesamiento de datos.
- Usuarios ingenuos: interactúan con el sistema invocando a uno de los programas de aplicación permanentes que se han escrito anteriormente.

Estructura del sistema global

Un sistema de base de datos se divide en módulos que tratan cada una de las responsabilidades del sistema general.

Los componentes funcionales de un sistema de base de datos incluyen:

- Gestor de archivos: gestiona la asignación de espacio en la memoria del disco y de las estructuras de datos usadas para representar información almacenada en disco.
- Gestor de base de datos: proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y las consultas que se hacen al sistema.
- Procesador de consultas: traduce sentencias en un lenguaje de consultas a instrucciones de bajo nivel que entiende el gestor de la base de datos. Además, intenta transformar una pregunta del usuario en forma equivalente pero más eficiente.
- Precompilador de DML: convierte las sentencias en DML incorporadas en un programa de aplicación en llamadas normales a procedimientos en el lenguaje principal. El precompilador debe interaccionar con el procesador de consultas para generar el código apropiado.
- Compilador de DDL: convierte sentencias en DDL en un conjunto de tablas que contienen metadatos.

Además, se requieren varias estructuras de datos como parte de la implementación del sistema físico, incluyendo:

- Archivos de datos: almacenan la base de datos.
- Diccionario de datos: almacena metadatos sobre la estructura de la base de datos.
- Indices: proporcionan acceso rápido a los elementos de datos que contienen valores determinados.

Unidad 2

Modelado de datos

El diseño de una base de datos es un proceso complejo que abarca varias decisiones a niveles muy distintos. La complejidad se controla mejor si se descompone el problema en subproblemas y se resuelven cada uno de estos independientemente, usando métodos y técnicas específicas. El diseño de base de datos se descompone en: diseño conceptual, diseño lógico y diseño físico.

Diseño conceptual

Parte de la especificación de requerimientos y el resultado es el esquema conceptual de la base de datos. Un esquema conceptual es una descripción de alto nivel de la estructura de la base de datos, independientemente del software de DBMS que se use para manipularla.

Diseño lógico

Parte del esquema conceptual y da como resultado un esquema lógico. Un esquema lógico es una descripción de la estructura de base de datos que puede procesar el software de DBMS. El diseño lógico depende de la clase de modelo de datos usado por el DBMS, no del DBMS utilizado.

Diseño físico

Parte del esquema lógico y da como resultado un esquema físico. El cual es la descripción de la implementación de una base de datos en la memoria secundaria, describe las estructuras de almacenamiento y los métodos usados para tener un acceso efectivo a los datos. Por esto, el diseño físico se adapta a un sistema DBMS específico. Hay una retroalimentación entre el diseño físico y lógico, porque las decisiones tomadas durante el diseño físico para mejorar el rendimiento pueden afectar la estructura del esquema lógico.

Modelo Entidad-Relación

Se basa en una percepción del mundo real que consiste en un conjunto de objetos básicos llamados entidades y relaciones entre estos objetos.

Una **entidad** es un objeto que existe y es distinguible de otros objetos. Una entidad puede ser concreta, tal como una persona o un libro, o puede ser abstracta, como un día festivo o un concepto.

Un **conjunto de entidades** es un conjunto de entidades del mismo tipo. Los conjuntos de entidades no necesitan ser disjuntos.

Una entidad está representada por un conjunto de atributos. Para cada atributo hay un conjunto de valores permitidos, llamados dominio de ese atributo.

Un **atributo** es una función que asigna un conjunto de entidades a un dominio. Así, cada entidad se describe por medio de un conjunto de pares (atributo, valor del dato), un par para cada atributo del conjunto de entidades.

Una base de datos incluye una colección de conjuntos de entidades cada uno de los cuales contiene un número cualquiera de entidades del mismo tipo.

Una **relación** es una asociación entre varias entidades.

Un **conjunto de relaciones** es un conjunto de relaciones del mismo tipo. Es una relación matemática de $n \geq 2$ conjuntos de entidades (posiblemente no distintos). Si E_1, E_2, .., E_n son conjuntos de entidades, entonces un conjunto de relaciones R es un subconjunto de $\{(e_1, e_2, .., e_n) | e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$ donde $(e_1, e_2, .., e_n)$ es una relación.

La función que juega una entidad en una relación se llama papel. Los papeles, normalmente son implícitos y no se suelen especificar. Sin embargo, son útiles cuando el significado de una relación necesita ser clarificado.

Puesto que la noción de conjunto de entidades y conjunto de relaciones no es precisa, es posible definir un conjunto de entidades y sus relaciones de varias formas diferentes. La principal diferencia está en la forma en que tratamos los diversos atributos.

Restricciones de asignación

Una restricción importante es la de las **cardinalidades de asignación**, que expresan el número de entidades con las que puede asociarse otra entidad mediante un conjunto de relaciones.

Para un conjunto binario de relaciones R entre los conjuntos de entidades A y B, la cardinalidad de asignación debe ser una de las siguientes:

- Una a una: una entidad en A está asociada a lo sumo con una entidad en B, y una entidad en B está asociada a lo sumo con una entidad en A.
- Una a muchas: una entidad en A está asociada con un número cualquiera de entidades en B. Una entidad en B, sin embargo, esta asociada a lo sumo con una entidad en A.
- Muchas a una: una entidad en A está asociada a lo sumo con una entidad en B. Una entidad en B, sin embargo, puede estar asociada con un número cualquiera de entidades en A.
- Muchas a muchas: una entidad en A está asociada con un número cualquiera de entidades en B, y una entidad en B está asociada con un número cualquiera de entidades en A.

La cardinalidad de asignación adecuada para un conjunto de relaciones determinado es dependiente del mundo real que el conjunto de relaciones está modelando.

Las **dependencias de existencia** constituyen otra clase importante de restricciones. Si la existencia de la entidad x depende de la existencia de la entidad y, entonces se dice que es **dependiente por existencia** de y. Esto significa que si se suprime y, también se suprime x. La entidad y se dice que es una **entidad dominante** y x se dice que es una **entidad subordinada**.

Claves

Una **superclave** es un conjunto de uno o más atributos que, considerados conjuntamente, nos permiten identificar de forma única a una entidad en el conjunto de entidades.

Una superclave puede contener atributos ajenos. Si K es una superclave, entonces también lo será cualquier superconjunto de K. A menudo estamos interesados en superclaves para las cuales ningún subconjunto propio es superclave. Dichas superclaves mínimas se llaman claves candidatas. Es posible que varios conjuntos de atributos distintos pudieran servir como **claves candidatas**.

Una **clave primaria** es una clave candidata que elige el diseñador de la base de datos como el medio principal de identificar entidades dentro de un conjunto de entidades.

El conjunto de entidades que no tiene atributos suficientes para formar una clave primaria se llama **conjunto de entidades débil**. El conjunto de entidades que tiene una clave primaria se denomina **conjunto de entidades fuerte**.

Un miembro de un conjunto de entidades fuerte es una **entidad dominante**, mientras que un miembro de un conjunto de entidades débil es una **entidad subordinada**.

Aunque un conjunto de entidades débil no tiene una clave primaria, debe haber un medio de distinguir entre todas aquellas entidades en el conjunto de entidades que dependen de una entidad fuerte determinada.

El **discriminador** de un conjunto de entidades débil es un conjunto de atributos que permite que se haga esta distinción.

La clave primaria de un conjunto de entidades débil está formada por la clave primaria del conjunto de entidades fuerte de la que dependen su existencia y su discriminador.

La clave primaria de un conjunto de entidades nos permite distinguir entre las diversas entidades del conjunto.

La composición de la clave primaria de un conjunto de relaciones R depende de la cardinalidad de asignación y de la estructura de los atributos asociados con el conjunto de relaciones R.

Si el conjunto de relaciones R no tiene atributos asociados, entonces el conjunto atributo(R) forma una superclave. Esta superclave es una clave primaria si la cardinalidad de asignación es muchas a muchas.

Si el conjunto de relaciones R tiene varios atributos asociados con él, entonces una superclave está formada con la posible adición de uno o más de estos atributos. La estructura de la clave primaria depende tanto de la cardinalidad como de las semánticas del conjunto de relaciones.

Diagrama de E-R

Consta de los siguientes componentes:

- Rectángulos, que representan conjuntos de entidades.
- Elipses, que representan atributos.
- Rombos, que representan relaciones entre conjuntos de entidades.
- Líneas, que conectan atributos a conjuntos de entidades y conjuntos de entidades a relaciones.

Cada componente se etiqueta con la entidad o relación que representa.

Un conjunto de entidades débil se indica en los diagramas E-R por medio de un rectángulo de doble contorno.

Conversiones al modelo lógico

Para cada conjunto de entidades y para cada conjunto de relaciones en la base de datos, existe una tabla única a la que se le asigna el nombre del conjunto de entidades o del conjunto de relaciones correspondiente. Cada tabla tiene un número de columnas que, a su vez, tienen nombres únicos.

Sea E un conjunto de entidades fuerte con los atributos descriptivos a_1, .., a_n. Representamos esta entidad por medio de una tabla llamada E con n columnas distintas cada una de las cuales corresponde a uno de los n atributos de E. Cada fila de esta tabla corresponde a una entidad del conjunto de entidades E.

Sea A un conjunto de entidades débil con atributos a_1, .., a_r. Sea B el conjunto de entidades fuerte del que depende A. La clave primaria de B consta de los atributos b_1, .., b_s. Representamos el conjunto de entidades A mediante una tabla llamada A con una columna para cada atributo del conjunto $\{a_1, .., a_r\}$ U $\{b_1,.., b_s\}$.

Sea R un conjunto de relaciones que implica a los conjuntos de entidades E_1, .., E_m. Supongamos que atributo(R) consta de n atributos. Representamos este conjunto de relaciones mediante una tabla llamada R con n columnas distintas, cada una de las cuales corresponde a uno de los atributos de atributo(R).

En general, la tabla para el conjunto de relaciones que conecta un conjunto de entidades débil con su correspondiente conjunto de entidades fuerte es redundante y no necesita presentarse en una representación tabular de un diagrama E-R.

Clasificación

Se usa para definir un concepto como una clase de objetos de la realidad caracterizado por propiedades comunes.

Generalización

Es una relación de inclusión que existe entre un conjunto de entidades de nivel más alto y uno o más conjuntos de entidades de nivel más bajo. Por ejemplo, cuenta (nivel más alto) es una generalización de cuenta-ahorros y cuenta-cheques (nivel más bajo).

La generalización se representa por medio de un triángulo etiquetado ISA. La etiqueta ISA significa "is a" (es un/a) y representa por ejemplo que una cuenta de ahorros "es una" cuenta.

La generalización se usa para hacer resaltar los parecidos entre tipos de entidades de nivel más bajo y ocultar sus diferencias. La distinción se hace a través de la herencia de atributos. Los atributos de los conjuntos de entidades de nivel más alto se dice que son heredados por los conjuntos de entidades de nivel más bajo.

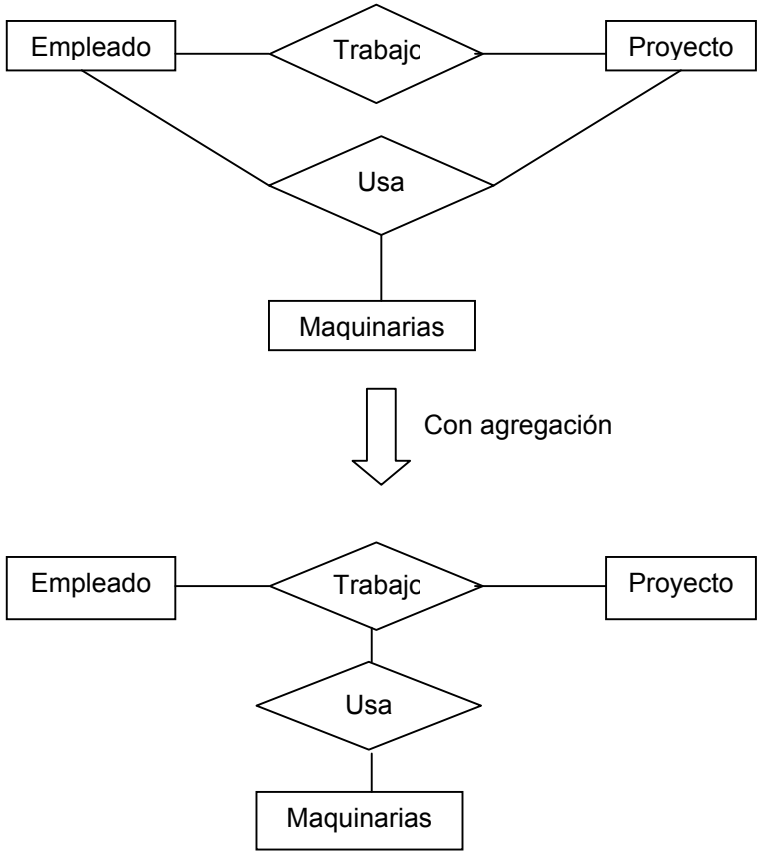
Existen dos métodos para transformar la generalización en una forma tabular:

1. Crear una tabla para el conjunto de entidades de nivel más alto. Para cada conjunto de entidades de nivel más bajo, crear una tabla que incluya una columna por cada uno de los atributos de ese conjunto de entidades más una columna por cada atributo de la clave primaria del conjunto de entidades de nivel más alto.
2. No crear una tabla para el conjunto de entidades de nivel más alto. En cambio, para cada conjunto de entidades de nivel más bajo, crear una tabla que incluya una columna para cada uno de los atributos de ese conjunto de entidades más una columna para cada atributo del conjunto de entidades del nivel más alto.

Agregación

La agregación es una abstracción a través de la cual las relaciones se tratan como entidades de nivel más alto. Un conjunto de entidades de este tipo se trata de la misma forma que cualquier otro conjunto de entidades.

La transformación de un diagrama E-R que incluya agregación a una forma tabular es directa.



Diseño de un esquema de base de datos E-R

Un diseñador de base de datos puede elegir entre una amplia variedad de alternativas. Entre las decisiones a tomar se encuentran:

- El uso de una relación ternaria o de un par de relaciones binarias.
- Si un concepto del mundo real se expresa mejor mediante un conjunto de entidades o por un conjunto de relaciones.
- El uso de un atributo o de un conjunto de entidades.
- El uso de un conjunto de entidades fuerte o débil.
- La oportunidad de usar generalización.
- La oportunidad de usar agregación.

Para tomar estas decisiones el diseñador de la base de datos necesita tener una buena comprensión de la empresa que va a modelar.

Dependencia funcional

Dada una relación R, el atributo Y de R **depende funcionalmente** del atributo X de R si y sólo si, un solo valor Y en R está asociado a cada valor X en R (en cualquier momento dado). Los atributos X e Y pueden ser compuestos.

Si el atributo X es una clave candidata de la relación R, entonces todos los atributos Y de la relación R deben por fuerza depender funcionalmente de X.

Se dice que el atributo Y de la relación R es por **completo dependiente funcionalmente** del atributo X de la relación R si depende funcionalmente de X y no depende funcionalmente de ningún subconjunto propio de X.

Si Y depende funcionalmente de X, pero no por completo, X debe ser compuesto.

Una dependencia funcional A → B se denomina **dependencia parcial** si existe un superconjunto propio y de A tal que y → B. Decimos que B depende parcialmente de y.

Una dependencia es **transitiva** si un atributo B (no clave) depende de un atributo C no clave.

Dada una relación R con atributos A, B, C decimos que existe una dependencia entre A y B multivaluada en R, si y sólo si el conjunto de valores de B que concuerdan con el par (A, C) en R depende solo del valor de A y es independiente del valor de C. Una **dependencia multivaluada** existe cuando un atributo puede determinar más de un valor para otro atributo.

Formas normales

Una relación está en 1º forma normal (**1NF**) si y sólo si todos los atributos tienen cardinalidad 0 o 1.

Una relación esta en 2º forma normal (**2NF**) si y sólo si está en 1NF y todos los atributos no clave dependen por completo de la clave primaria, es decir, no existen dependencias parciales.

Una relación está en 3º forma normal (**3NF**) si y sólo si está en 2NF y todos los atributos no clave dependen de manera no transitiva de la clave primaria.

Una relación está en **forma normal Boyce Codd** (BCNF) si y sólo si todo determinante es una clave candidata. Un determinante es un atributo del cual depende funcionalmente (por completo) algún otro atributo.

Una relación está en 4º forma normal (**4NF**) si está en BCNF y no existen dependencias multivaluadas.

Una relación está en 5º forma normal (**5NF**) si cada restricción es una consecuencia lógica de la definición de las claves y dominios

Normalización

La **normalización** de relaciones es un proceso de diseño que toma un conjunto de relaciones y genera otro que preserva la semántica, minimiza la redundancia y elimina las dependencias funcionales, transitivas y Boyce Codd. El proceso está guiado por las dependencias funcionales.

Unidad 3

Modelo relacional

Una base de datos relacional consiste en una colección de tablas (relaciones), a cada una de las cuales se asigna un nombre único. Una fila (tupla) de una tabla representa una relación entre un conjunto de valores.

Definimos una **relación** como un subconjunto de un producto cartesiano de una lista de dominios.

Para cada atributo hay un conjunto de valores permitidos, llamado **dominio** (D_i), de ese atributo. En general, una tabla de n columnas debe ser un subconjunto de $D_1 \times D_2 \times ... \times D_{n-1} \times D_n$. Un dominio es atómico si los elementos del dominio se consideran unidades indivisibles.

Restricciones de integridad

Existen tres tipos de restricciones que se cumplen automáticamente: restricciones de clave, restricciones de integridad de entidades, restricciones de integridad referencial. Las **restricciones de clave** indican que todas aquellas claves candidatas no pueden tener valores repetidos. Las **restricciones de integridad de entidades** establecen que ningún valor de clave primaria puede ser nulo. Esto es porque el valor de la clave primaria se usa para identificar las tuplas individuales de una relación; permitir valores nulos para la clave primaria implica que no se pueda identificar algunas tuplas. Las **restricciones de integridad referencial** se especifica entre dos relaciones, y se usa para mantener la congruencia entre las tuplas de dos relaciones (establece que una tupla de una relación que haga referencia a otra relación debe referirse a una tupla existente en esa relación).

Selección de la clave primaria

La mayoría de los DBMS requieren que se escoja como clave primaria uno de los identificadores de una entidad. Esta clave sirve como identificador para hallar un caso único de la entidad, dado el valor de la clave primaria. Para elegir un identificador se pueden tener en cuenta dos criterios:

1. Seleccionar como clave primaria el identificador usado para accesos directos por el máximo número de operaciones. Si una entidad tiene múltiples identificadores, se debe designar uno de ellos como clave primaria de la entidad.
2. Preferir los identificadores simples a los múltiples y los internos a los externos; de esta manera, las claves primarias de las entidades se pueden mantener mínimas en tamaño y simples en estructura.

Un identificador (clave) es **simple** si esta formado por un solo atributo o entidad, es **compuesto** si tiene más de uno.

Un identificador es **interno** si no se utilizan entidades relacionadas en su conformación, es **externo** en caso contrario.

Un identificador es **mixto** si está compuesto por atributos y entidades relacionadas.

Conversiones del modelo E-R lógico al modelo relacional.

- Eliminación de identificadores externos: se deben transformar los identificadores externos en internos. Esto se logra agregando el identificador externo como interno en la entidad o relación.
- Eliminación de atributos compuestos y polivalentes: Con cada atributo compuesto, se tienen dos alternativas:
    1) Eliminar el atributo compuesto considerando todos sus componentes como atributos individuales.
    2) Eliminar los componentes individuales y considerar cada atributo compuesto entero como un solo atributo.
- Transformación de entidades: se transforma cada entidad en una relación. Los atributos y la clave primaria de la entidad se convierten en los atributos y clave primaria de la relación.

- Transformación de interrelaciones de uno a uno: existen dos posibles casos:
  1) la participación de las dos entidades en la interrelación es total.
  2) una de las dos o las dos tienen participación parcial.

  En el primer caso si se da que las dos entidades tienen igual clave primaria, las dos relaciones correspondientes se integran a una relación combinando todos los atributos e incluyendo la clave primaria sólo una vez.

  El primer caso puede darse con dos entidades con clave primaria diferente, se procede igual, juntando las entidades y se toma como clave primaria una de las dos.

  El segundo caso tiene también dos posibilidades. Una de ellas con participación parcial de una entidad, puede ocurrir que se solucione como el caso anterior poniendo la clave adecuada, dado que al traer los atributos debe dejarse la posibilidad de que tengan valores nulos y por lo tanto esa otra clave no está disponible. Otra solución es crear tres relaciones una por cada entidad y otra por la interrelación, la clave primaria de la interrelación está dada por cualquiera de las dos que la forman.

  Si las dos entidades tienen participación parcial, para evitar valores nulos y representar tanto las entidades como la interrelación, se crea una relación proveniente de la interrelación.
- Transformación de interrelaciones de uno a muchos: Si la entidad del lado de muchos tiene una participación obligatoria, se crean dos relaciones una por cada entidad y la relación proveniente de convertir la "una" recibe la clave primaria de la de muchos.

  Si la entidad del lado de muchos tiene una participación parcial, hay dos soluciones. Si se permiten valores nulos se puede construir dos relaciones, una por cada entidad, y la "una" trae la llave de la muchos. La otra posibilidad es crear tres relaciones, poniendo una por la interrelación.
- Transformación de interrelaciones de muchos a muchos: la conversión se realiza poniendo una relación por cada entidad y otra por la interrelación.
- Transformación de interrelaciones n-arias y recursivas: cada entidad se transforma en una relación y la interrelación también, la clave primaria está dada por la concatenación de cada clave de cada entidad.

  Si la interrelación es recursiva:
  - ✓ Uno a muchos: cualquiera de las dos soluciones.
  - ✓ Muchos a muchos: la segunda solución.
  - ✓ Uno a uno: hay que analizar cual es la mejor.
- Eliminación de jerarquías de generalización: para eliminar esta estructura existen 3 opciones dependientes de las características de parcial/total y exclusiva/superpuesta. Las tres variantes pueden ser:
  - ✓ Eliminar las especializaciones, incluyendo en la generalización los atributos de las mismas y además un nuevo atributo que indicará el tipo de la ex-generalización.
  - ✓ Eliminar la generalización y dejar las especializaciones. Esta variante es posible solo si la jerarquía es total, porque sino quedarán elementos sin ser representados por alguna entidad.
  - ✓ Retener tanto especializaciones como generalizaciones, incluyendo por cada especialización una interrelación "es un" como la generalización.

Unidad 4

Lenguajes de consulta

Un lenguaje de consulta es un lenguaje en el que un usuario solicita información de la base de datos. Los lenguajes de consulta pueden clasificarse en lenguajes procedimentales o no procedimentales. En un **lenguaje procedimental**, el usuario da instrucciones al sistema para que realice una secuencia de operaciones en la base de datos para calcular el resultado deseado. En un **lenguaje no procedimental**, el usuario describe la información deseada sin dar un procedimiento específico para obtener esa información.

Algebra relacional

El **álgebra relacional** es un lenguaje de consulta procedimental. Consta de un conjunto de operaciones que toman una o dos relaciones como entrada y producen una nueva relación como resultado. Las operaciones fundamentales en el álgebra relacional son: seleccionar, proyectar, producto cartesiano, renombrar, unión y diferencia de conjuntos. Además de las operaciones fundamentales existen otras operaciones: intersección de conjuntos, producto natural, división y asignación.

Operaciones fundamentales

Las operaciones seleccionar, proyectar y renombrar se llaman operaciones unitarias, ya que operan sobre una relación. Las otras tres operaciones operan sobre pares de relaciones y, por lo tanto, se llaman operaciones binarias.

- Seleccionar($\sigma_{predicado}$): Selecciona tuplas que satisfacen un predicado dado.
- Proyectar ($\pi_{atributos}$): Devuelve la relación argumento con columnas omitidas.
- Producto cartesiano (x): conecta dos entidades de acuerdo a la definición matemática de la operación.
- Renombrar(ρ_x(r)): Permite utilizar la misma tabla en un, por ejemplo, producto cartesiano.
- Unión (U): tuplas comunes a dos relaciones, equivalente a la unión matemática. Debe efectuarse entre relaciones con sentido.
- Diferencia de conjuntos (-): Permite encontrar tuplas que estén en una relación pero no en otra. La expresión r - s da como resultado una relación que contiene aquellas tuplas que están en r pero no en s.

Definición formal

Una expresión básica en el álgebra relacional consta de cualquiera de las siguientes:

- Una relación en la base de datos.
- Una relación constante.

Una expresión general en el álgebra relacional se construye a partir de subexpresiones. Sean E_1 y E_2 expresiones del álgebra relacional. Entonces las siguientes son todas expresiones del álgebra relacional:

- E_1 U E_2.
- E_1 - E_2.
- E_1 x E_2.
- σ_P(E1), donde P es un predicado con atributos de E_1.
- π_S(E1), donde S es una lista que consta de algunos de los atributos de E_1.
- ρ_X(E_1), donde X es el nuevo nombre de la relación E_1.

Operaciones adicionales

Las operaciones adicionales no añaden ninguna potencia al álgebra, pero simplifican consultas comunes.

- Intersección ($\cap$): equivale a la intersección matemática. $R \cap S = R - (R-S)$

- Producto natural (|x|): hace el producto cartesiano con una selección de tuplas "con sentido" eliminando las columnas (atributos) repetidas. Si R y S son dos relaciones que no tienen atributos en común, es igual al producto cartesiano.

  $R \,|X|\, S = \pi_{R\ U\ S} (\sigma_{R.A1=S.A1\ \wedge\ ...\ \wedge R.An = S.An} (RxS))$
- División (÷): se establece para aquellas consultas que incluyen la frase "para todos". Sean r(R) y s(S) relaciones, y S esta incluido en R. La relación r ÷ s es una relación de esquema R - S. Una tupla t está en r ÷ s si para cada tupla t_s en s existe una tupla t_r en r que satisface las dos condiciones siguientes:

  $$t_r[S] = t_s[S]$$
  $$t_r[R - S] = t\,[R - S]$$

  $r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - r)$
- Asignación (⇐): expresión que asigna a una variable temporal el resultado de una operación.

Cálculo relacional de tuplas

El **cálculo relacional de tuplas**, es un lenguaje de consultas no procedimental. Describe la información deseada sin dar un procedimiento específico para obtener esa información..

Una consulta en el cálculo relacional de tuplas se expresa como {t|P(t)} es decir, el conjunto de todas las tuplas t, tal que el predicado P, es verdadero para t.

Una **fórmula** en el cálculo relacional de tuplas se compone de átomos. Un **átomo** tiene una de las siguientes formas:

- $s \in r$, donde s es una variable de tupla y r es una relación.
- $s[x]\ \Theta\ u[y]$, donde s y u son variables de tuplas, x es un atributo sobre el que s está definida, y es un atributo sobre el que u está definida, y Θ es un operador de comparación (<, ≤, =, >, ≥). Se requiere que los atributos x e y tengan dominios cuyos miembros puedan compararse por medio de Θ.
- $s[x]\ \Theta\ c$, donde s es una variable de tupla, x es un atributo sobre el que s está definida, Θ es un operador de comparación, y c es una constante en el dominio del atributo x.

Las fórmulas se construyen a partir de los átomos usando las siguientes reglas:

- Un átomo es una fórmula.
- Si P_1 es una fórmula, entonces también lo es $\sim P_1$.
- Si P_1 y P_2 son fórmulas, entonces también lo son $P_1 \vee P_2$, $P_1 \wedge P_2$, $P_1 \Rightarrow P_2$.
- Si $P_1(s)$ es una fórmula que contiene una variable de tupla libre s, entonces $\forall\ s \in r(P_1(s))$ y $\exists\ s \in r(P_1(s))$ también son fórmulas.

Seguridad de expresiones

Una expresión en el cálculo relacional de tuplas puede generar una relación infinita.

El **dominio** de P, representado por dom(P), es el conjunto de todos los valores referenciados por P. Estos incluyen a los valores mencionados en P, así como a los valores que aparecen en una tupla de una relación mencionada en P. Así, el dominio de P es el conjunto de todos los valores que aparecen en una o más relaciones cuyos nombres aparecen en P.

Decimos que una expresión {t|P(t)} es **segura** si todos los valores que aparecen en el resultado son valores de dom(P).

El cálculo relacional de tuplas restringido a expresiones seguras es equivalente en poder expresivo al álgebra relacional.

Cálculo relacional de dominios

Se utilizan variables de dominio que toman valores del dominio de un atributo en lugar de tuplas completas.

Una **expresión** en el cálculo relacional de dominios es de la forma $\{<x_1, x_2, .., x_n> \mid P(x_1, x_2, .., x_n)\}$, donde $x_1, x_2, .., x_n$ representan variables de dominio. P representa una **fórmula** compuesta por átomos. Un **átomo** en el cálculo relacional de dominios tiene una de las formas siguientes:

- $<x_1, x_2,.., x_n> \in r$, donde r es una relación en n atributos y $x_1, x_2,.., x_n$ son variables de dominio o constantes de dominio.
- $x \Theta y$, donde x e y son variables de dominio y Θ es un operador de comparación ($<, \leq, =, \neq, >, \geq$). Es requisito que los atributos x e y tengan dominios que puedan compararse por medio de Θ.
- $x \Theta c$, donde x es una variable de dominio, Θ es un operador de comparación y c es una constante en el dominio del atributo para el cual x es una variable de dominio.

Las fórmulas se construyen a partir de los átomos usando las siguientes reglas:

- Un átomo es una fórmula.
- Si P_1 es una fórmula, entonces también lo es $\sim P_1$.
- Si P_1 y P_2 son fórmulas, entonces también lo son $P_1 \vee P_2$, $P_1 \wedge P_2$, $P_1 \Rightarrow P_2$.
- Si $P_1(x)$ es una fórmula en x, donde x es una variable de domino, entonces $\forall x (P_1(x))$ y $\exists x (P_1(x))$ también son fórmulas.

En el cálculo relacional de tuplas, cuando escribimos $\exists s$ para alguna variable de tupla s, lo asociamos directamente a una relación escribiendo $\exists s \in r$. Sin embargo, cuando escribimos $\exists b$ en el cálculo relacional de dominios, b no está relacionado con una tupla, sino a un valor de dominio.

Seguridad en las expresiones

Decimos que una expresión $\{<x_1, x_2,.., x_n> \mid P(x_1, x_2,.., x_n)\}$ es **segura** si se cumplen todas las condiciones siguientes:

- Todos los valores que aparecen en tuplas de la expresión son valores de dom(P).
- Para cada subfórmula "existe" de la forma $\exists x (P_1(x))$, la subfórmula es verdadera si y solo si, existe un valor x en $dom(P_1)$ tal que $P_1(x)$ es verdadero.
- Para cada subfórmula "para todos" de la forma $\forall x (P_1(x))$, la subfórmula es verdadera si y solo si, $P_1(x)$ es verdadera para todos los valores x de $dom(P_1)$.

Cuando el cálculo relacional de dominios se restringe a expresiones seguras, es equivalente al cálculo relacional de tuplas restringido a expresiones seguras.

Operaciones de Updates (solo para AR)

Inserción

$r \Leftarrow r \cup E$ (r relación y E nueva tupla)

Eliminación

$r \Leftarrow r - E$

Actualización de datos

$\delta_{A \Leftarrow E}$ (r)
Ej: $\delta_{saldo \Leftarrow saldo * 1.05}$ (depósito)

Optimización de consultas

La ejecución real de una consulta implicará muchos accesos a disco.

Dada una consulta, generalmente existe una variedad de métodos para calcular la respuesta. Es responsabilidad del sistema transformar la consulta hecha por el usuario en una consulta equivalente que pueda calcularse de manera más eficiente. Esta "optimización" o mejora de la estrategia para procesar una consulta se denomina **optimización de consultas**.

La primera acción que debe tomar el sistema con una consulta es traducirla a su forma interna, la cual normalmente está basada en el álgebra relacional. En el proceso de generar la forma interna de la consulta, el analizador sintáctico (parser) comprueba la sintaxis de la consulta del usuario, verifica que los nombres de la relación que aparecen en la consulta sean nombres de la base de datos, y así sucesivamente. Si la consulta se expresó en términos de una vista, el parser sustituye todas las referencias al nombre de la vista por la expresión del álgebra relacional para calcular la vista.

Cada expresión del álgebra relacional representa una secuencia determinada de operaciones. El primer paso para seleccionar una estrategia de procesamiento de consultas es encontrar una expresión en álgebra relacional que sea equivalente a la dada y se ejecute de manera más eficiente.

Expresiones equivalentes:

- Hacer las selecciones lo antes posible. Cambiar $\sigma_{p1 \wedge p2}(e) \rightarrow \sigma_{p1}(\sigma_{p2}(e))$.
- Resolver la proyección lo antes posible.
- Producto natural: (r1 |x| r2 |x| r3) = r1 |x| (r2 |x| r3).
- $\sigma_p(r1 \cup r2) = \sigma_p(r1) \cup \sigma_p(r2)$.
- $\sigma_p(r1 - r2) = \sigma_p(r1) \cup \sigma_p(r2)$.
- $(r1 \cup r2) \cup r3 = r1 \cup (r2 \cup r3)$.
- r1 |x| r2 = r2 |x| r1.

Estimación del costo de las consultas

Para poder elegir una estrategia basada en información fiable, los sistemas de base de datos pueden almacenar estadísticas por cada relación r. Estas estadísticas incluyen:

- El número de tuplas en la relación (nr).
- El tamaño en bytes de la tupla (sr).
- El número de valores distintos que aparecen en la relación r para un atributo determinado V(a, r).

Costos de las consultas:

- Producto cartesiano: r x t
  # tuplas nr * nt
  # bytes en cada tupla sr + st
- Selección: σ_p (r)
  # tuplas $\frac{nr}{V(A, r)}$ se supone distribución uniforme
  # bytes de la tupla sr
- Producto natural: r |x| t
  Tres casos: r1(R1), r2(R2) →
  $R1 \cap R2 = \varnothing \rightarrow$ r1 |x| r2 = r1 x r2
  $R1 \cap R2$ clave en R1 o R2, C/tupla de r2 une una tupla de r1 (o viceversa) → r1 |x| r2 = r2
  $R1 \cap R2$ no es clave → $R1 \cap R2 = \{ A \}$
  cada $t \in r1$ producto $\frac{nr2}{V(A,r2)}$ tuplas r1 |x| r2
  → # tuplas $\frac{nr1 * nr2}{V(A, r2)}$
  Idem con $t \in r2 \rightarrow \frac{nr2 * nr1}{V(A, r1)}$
  Tomo el menor

La información estadística respecto de las relaciones es útil cuando hay varios índices disponibles para facilitar el procesamiento de una consulta. La presencia de las estructuras tiene una influencia considerable sobre la elección de la estrategia de procesamiento de una consulta.

Las consultas que implican un producto natural pueden procesarse de varias maneras, dependiendo de la disponibilidad de índices y de la forma de almacenamiento físico que se utilizó para cada relación. Si las tuplas de una relación se almacenan juntos físicamente, puede ser conveniente una estrategia de intersección orientada a bloques. Si las relaciones están ordenadas, puede ser deseable una intersección - combinación.

En un sistema de multiprocesadores, las intersecciones se pueden calcular de manera eficiente dividiendo la tarea entre varios procesadores. En una máquina de memoria compartida es posible lograr paralelismo dividiendo una de las dos relaciones que van a participar en la intersección y procesando cada partición en paralelo.

Unidad 5

SQL

Es el lenguaje de bases de datos relacionales estándar. Tiene las siguientes partes:

1) Lenguaje de definición de datos (DDL): proporciona ordenes para definir esquemas de relación, eliminar relaciones, crear índices y modificar esquemas de relación.
2) Lenguaje de manipulación de datos interactivo: incluye un lenguaje de consultas basado en el álgebra relacional y el cálculo relacional de tuplas. También incluye ordenes para insertar, suprimir y modificar tuplas de la base de datos.
3) Lenguaje de manipulación de datos inmerso (DML): está diseñado para usar lenguajes de programación de propósito general.
4) Definición de vista.
5) Autorizaciones al acceso a datos.
6) Integridad: restricciones complejas.
7) Control de transacciones: incluye ordenes para especificar el comienzo y el final de las transacciones. Permite el bloqueo explícito de los datos para control de concurrencia.

Estructura básica

La estructura básica de una expresión en SQL consta de tres cláusulas: **select**, **from** y **where**.

La cláusula **select** corresponde a la operación de proyección del álgebra relacional. Se usa para listar los atributos que se desean en el resultado de una consulta.

La cláusula **from** corresponde a la operación de producto cartesiano del álgebra relacional. Lista las relaciones que se van a examinar en la evaluación de la expresión.

La cláusula **where** corresponde al predicado de selección del álgebra relacional. Consta de un predicado que implica atributos de las relaciones que aparecen en la cláusula from.

$\pi_{a1,...,an} (\sigma_p (r_1 \times ... \times r_m)$) equivale a

```
Select   a1,..., an
From     r1,..., rm
Where    P
```

Si se omite la cláusula where, el predicado P es verdadero. La lista de atributos a1,..., an puede sustituirse por un asterisco (*) para seleccionar todos los atributos de todas las relaciones que aparecen en la cláusula from.

Operaciones de conjuntos y tuplas duplicadas

SQL permite duplicados en las relaciones. En aquellos casos en los que queremos forzar la eliminación de duplicados, insertamos la palabra clave **distinct** después de select.

Con la palabra clave **all** especificamos explícitamente que no se eliminan los duplicados.

SQL incluye las operaciones **union** (unión), **intersect** (intersección) y **minus** (diferencia), que operan sobre relaciones y corresponden a las operaciones del álgebra relacional unión, intersección y diferencia.

Por defecto, la operación union elimina las tuplas duplicadas. Para retener duplicados se debe escribir union all en lugar de union.

La cláusula **unique** devuelve verdadero si la subconsulta que se pasa como argumento no produce tuplas duplicadas.

Predicados y conectores

SQL usa los conectores lógicos **and**, **or** y **not** en vez de los símbolos matemáticos. Permite el uso de expresiones aritméticas como operandos de los operadores de comparación. Una expresión aritmética puede implicar cualquiera de los operadores, +, -, * y /, operando sobre constantes o valores de tuplas.

También incluye un operador de comparación **between** para simplificar cláusulas where que especifican un valor que sea menor o igual que un valor dado y mayor o igual que otro valor dado. También se puede usar el operador **not between**.

También incluye un operador de selección para comparaciones de cadenas de caracteres. Los modelos se describen usando dos caracteres especiales:

- **%**: este carácter es igual a cualquier subcadena.
- _ : es igual a cualquier carácter.

Estos son sensibles a mayúsculas y minúsculas.

Los modelos se expresan usando el operador de comparación **like**.

Pertenencia a un conjunto

El conector **in** prueba la pertenencia a un conjunto, donde el conjunto es una colección de valores producidos por una cláusula select. El conector **not in** prueba la no pertenencia al conjunto.

La cláusula se puede anidar.

Comparación de conjuntos

La frase "mayor que algún" se representa por >**some**. También se permite las comparaciones <some, $\leq$ some, $\geq$ some, = some y ≠some.

La construcción > all corresponde a la frase "mayor que todos". Como en el caso de some, SQL permite las comparaciones < all, $\leq$ all, $\geq$ all, = all y ≠all.

Puesto que un select genera un conjunto de tuplas, a veces podemos querer comparar conjuntos para determinar si un conjunto contiene todos los miembros de algún otro conjunto. Tales comparaciones se hacen usando las construcciones **contains** y **not contains**.

Pruebas para relaciones vacías

La construcción **exists** devuelve el valor true si la subconsulta del argumento no está vacía.

La no existencia de tuplas en una subconsulta puede probarse usando la construcción **not exists**.

Ordenación de la presentación de tuplas

SQL ofrece al usuario cierto control sobre el orden en el que se van a presentar las tuplas en una relación. La cláusula **order by** hace que devuelva las tuplas en el resultado de una consulta en un orden determinado. Por defecto, SQL lista los elementos en orden ascendente. Para especificar el tipo de ordenación, podemos especificar **desc** para descendente o **asc** para orden ascendente. Además, el orden puede realizarse sobre múltiples atributos.

La cláusula **having** permite aplicar condiciones a los grupos. Si en la misma consulta aparecen una cláusula where y una cláusula having, primero se aplica el predicado de la cláusula where. Las tuplas que satisfacen el predicado where son colocadas en grupos por la cláusula group by. Después se aplica la cláusula having a cada grupo. Los grupos que satisfacen el predicado de la cláusula having son utilizados por la cláusula select para generar tuplas del resultado de la consulta. Si no hay cláusula having, el conjunto completo de tuplas que satisfacen la cláusula where se trata como un grupo único.

Funciones de agregación

SQL ofrece la posibilidad de calcular funciones en grupos de tuplas usando la cláusula **group by**. El atributo o atributos dados en la cláusula group by se usan para formar grupos. Las tuplas con el mismo valor en todos los atributos en la cláusula group by se colocan en un grupo. SQL incluye funciones para calcular:

- Promedio (**avg**).
- Mínimo (**min**).
- Máximo (**max**).
- Total (**sum**).
- Contar (**count**).

Las operaciones como avg se llaman funciones de agregación porque operan sobre grupos de tuplas. El resultado de una función de agregación es un valor único.

La función de agregación count se usa frecuentemente para contar el número de tuplas de una relación. La notación para esto es count(*).

Modificaciones de la base de datos

Eliminación

Se pueden suprimir solamente tuplas completas, no se pueden suprimir valores solo de atributos determinados. En SQL una supresión se expresa por medio de **delete r where P**, donde P representa un predicado y r una relación. Las tuplas t en r, para las cuales P(t) es verdadero, son eliminadas en r.

Si queremos eliminar tuplas de varias relaciones, debemos usar una orden delete para cada relación.

Inserción

Para insertar datos en una relación, especificamos una tupla que se va a insertar o escribimos una consulta cuyo resultado es un conjunto de tuplas que se van a insertar. Los valores de atributos para las tuplas insertadas deben ser miembros del dominio de los atributos y las tuplas insertadas deben tener el número correcto de atributos.

Una inserción en SQL se expresa por medio de **insert into** r **values** (v_1, v_2, .., v_n) donde r es una relación y (v_1, v_2, .., v_n) son los valores de los atributos de r.

Actualizaciones

En ciertas situaciones podemos desear cambiar un valor en una tupla sin cambiar todos los valores en la tupla. Para este propósito puede usarse la sentencia **update**. Como en el caso de insert y delete se puede elegir las tuplas que se van a actualizar usando una consulta.

La forma general de la sentencia update es **update** r **set** expresión. En general, la cláusula update puede contener cualquier construcción legal en la cláusula where de la sentencia select.

Valores nulos

Es posible, que para las tuplas insertadas se den valores únicamente en algunos atributos del esquema. El resto de los valores son asignados a valores nulos representados por **null**.

Todas las comparaciones que implican null son falsas por definición. Sin embargo, la palabra clave null puede usarse en un predicado para probar si hay un valor nulo.

El predicado **is not null** prueba la ausencia de un valor nulo.

Vistas

Para definir una vista debemos dar a la vista un nombre y declarar la consulta que calcula la vista. La forma de la orden create view es **create view** v **as** <expresión de consulta> donde <expresión de consulta> es cualquier expresión de consulta permitida.

Unidad 6

Definición de transacción

Es una colección de operaciones que forman una única unidad lógica de trabajo.

Es una unidad de programa que accede y actualiza elementos de información. La consistencia de la base de datos se debe mantener antes y después de la ejecución de la transacción, aunque durante su ejecución se puede permitir una inconsistencia temporal que puede ocasionar dificultades si se produce algún fallo.

Las propiedades de las transacciones (ACID) son:

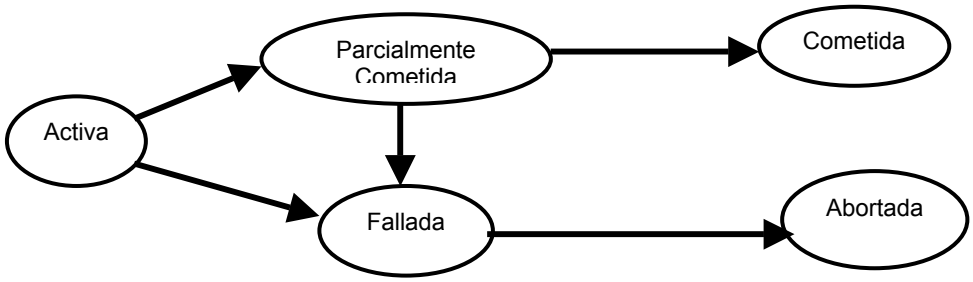
- Atomicidad: todas las operaciones asociadas a una transacción deben ejecutarse por completo o no ejecutarse.
- Consistencia: la ejecución aislada de la transacción conserva la consistencia de la base de datos.
- Aislamiento (isolation): cada transacción ignora el resto de las transacciones que se ejecutan concurrentemente en el sistema, actúa cada una como única.
- Durabilidad: una transacción terminada con éxito realiza cambios permanentes en la base de datos, incluso si hay fallos en el sistema.

Estados de una transacción

Una transacción debe estar en uno de los siguientes estados:

- Activo: estado inicial, estado normal durante la ejecución.
- Parcialmente cometido: después de que se haya ejecutado la última sentencia. Todavía es posible que tenga que abortarse, ya que puede que la salida real aún no se haya escrito en el disco, y por lo tanto, un fallo de hardware puede impedir la terminación con éxito.
- Fallado: después de descubrir que la ejecución normal no puede proceder. Una transacción así debe retroceder. Una vez que se ha efectuado el retroceso, la transacción entra en el estado abortado.
- Abortado: después de que la transacción haya retrocedido y se haya restaurado la base de datos al estado en que estaba antes de empezar la transacción. En este momento el sistema operativo tiene dos opciones:
  - ✓ Reiniciar la transacción: solo si la transacción se abortó como resultado de un error de hardware o software que no se haya debido a la lógica interna de la transacción. Una transacción reiniciada se considera una transacción nueva.
  - ✓ Eliminar la transacción: cuando se producen errores en la lógica interna que puede corregirse solo volviendo a escribir el programa, o debido a una mala entrada, o debido a que los datos deseados no se encontraron en la base de datos.
- Cometido: una transacción entra en estado cometido si se ha cometido parcialmente y se garantiza que nunca se abortará.

Diagrama de estados de una transacción



Modelo de transacción

```
read (A, a1)
a1 := a1 – 100;
write(A, a1)
read (B, b1)
b1 := b1 + 100;
write(B, b1)
```

Los movimientos de bloques entre el disco y la memoria principal se inician por medio de las dos operaciones siguientes:

- **input** (X), que transfiere el bloque físico en el que reside el elemento de información X a la memoria principal.
- **output** (X), que transfiere el bloque de registro intermedio en el que reside X al disco y sustituye el bloque físico apropiado que allí se encuentra.

Las transacciones interactúan con el sistema de base de datos transfiriendo los datos desde variables del programa a la base de datos y desde la base de datos a variables de programa. Esta trasferencia de datos se logra utilizando las dos operaciones siguientes:

- **read** (X, x_i), que asigna el valor del elemento de información X a la variable local x_i.
- **write** (X, x_i), que asigna el valor de la variable local x_i al elemento de información X en el bloque que está en el registro intermedio.

En las dos operaciones si el bloque en el que reside X no está en memoria principal, entonces se ejecuta input(X).

Transacciones concurrentes y transacciones distribuidas

Varias transacciones ejecutándose al mismo tiempo comparten el procesador, el sistema debe controlar la interacción entre las transacciones para conservar la consistencia de la base de datos. Las transacciones se ejecutan en orden, una a una T_0, T_1, .., T_n y para conservar la consistencia el resultado luego de la ejecución debe ser T_n, .., T_1, T_0.

En un sistema distribuido cada una de las localidades que lo componen pueden querer ejecutar las transacciones que acceden a datos de una localidad o de varias localidades.

En un sistema monousuario las transacciones pueden ejecutarse concurrentemente o no.

Una transacción es **local** cuando se accede a datos que están en la misma localidad.

Una transacción es **global** cuando los datos están en distintas localidades.

Las transacciones en un sistema monousuario son locales, mientras que en un sistema distribuido pueden ser tanto locales como globales.

En un sistema monousuario las actualizaciones son más rápidas que en un sistema centralizado y distribuido.

En los sistemas centralizados en general son más rápidas las actualizaciones que en un sistema distribuido, a causa de la repetición y/o fragmentación de la base de datos.

En un sistema distribuido pueden ejecutarse transacciones en paralelo, en los sistemas monousuario no.

Es más difícil garantizar la atomicidad en los sistemas distribuidos que en los sistemas centralizados.

Clasificación de fallos

El tipo de fallo más sencillo de tratar es el que no resulta en pérdida de información en el sistema. Los fallos más difíciles de tratar son los que resultan en perdida de información. Los algoritmos para asegurar la consistencia de la base de datos y la atomicidad de las transacciones a pesar de fallos tienen dos partes:

1) Acciones tomadas durante el procesamiento normal de la transacción para asegurar que existe suficiente información para permitir la recuperación de fallos.
2) Acciones tomadas a continuación de un fallo para asegurar la consistencia de la base de datos y la atomicidad de las transacciones.

Tipos de fallos

- Errores lógicos: la transacción no puede continuar con su ejecución normal debido a alguna condición interna, como puede ser una entrada inválida, información no localizada, desbordamiento o que se exceda el límite de los recursos.
- Errores del sistema: el sistema ha entrado en un estado no deseable, como resultado del cual la transacción no puede continuar con su ejecución normal. Sin embargo, la transacción puede volverse a ejecutar más tarde.
- Caída del sistema: el hardware funciona mal, causando la perdida del contenido del almacenamiento volátil. El contenido del almacenamiento no volátil permanece intacto.
- Fallo de disco: un bloque del disco pierde su contenido debido a la rotura de la cabeza o a fallos durante una operación de transacción de información.

Estructuras de almacenamiento

Almacenamiento volátil

La información que reside en memoria volátil normalmente no sobrevive a las caídas del sistema. El acceso a memoria volátil es muy rápido, debido a la velocidad del acceso a memoria ya que es posible acceder a cualquier dato en memoria volátil directamente.

Almacenamiento no volátil

La información que reside en memoria no volátil normalmente no sobrevive a las caídas del sistema. El almacenamiento no volátil es más lento que el volátil por varios ordenes de magnitud.

Almacenamiento estable

La información que reside en el almacenamiento estable nunca se pierde. Se debe repetir información en varios medios de almacenamiento no volátil independientes y actualizar la información de una manera controlada para asegurar que se mantienen los datos ante todo caso.

Recuperación basada en bitácora

Cuando ocurre un error durante la ejecución de una transacción, podemos invocar uno de los dos procedimientos de recuperación posibles:

- Volver a ejecutar la transacción.
- No volver a ejecutar la transacción.

En los dos casos la base de datos se deja en un estado inconsistente. La razón de esta dificultad es que hemos modificado la base de datos sin tener la seguridad de que la transacción se va a cometer realmente.

Para lograr el objetivo de atomicidad, primero debemos sacar información describiendo las modificaciones de almacenamiento estable sin modificar la base de datos. Esto permitirá sacar todas las modificaciones que hizo la transacción cometida a pesar de fallos.

La estructura más utilizada para grabar las modificaciones de la base de datos es la bitácora. Cada registro describe una única escritura de la base de datos y tiene los siguientes campos:

- Nombre de la transacción.
- Nombre del dato.
- Valor antiguo.
- Valor nuevo.

Existen otros registros de bitácora especiales para grabar sucesos importantes durante el procesamiento de la transacción, tales como el comienzo de una transacción y el cometido o aborto de una transacción.

- <T_i **star**> Inicio de transacción T_i.
- <T_i, X_j, V_1, V_2> La transacción T_i ha realizado una escritura en el dato X_j. X_j tenía el valor V_1 antes de la escritura y tendrá el valor V_2 después de la escritura.
- <T_i **commit**> La transacción T_i ha terminado.
- <T_i **abort**> La transacción T_i fue abortada.

Siempre que una transacción realice una escritura, es fundamental que se cree el registro de bitácora para esa escritura antes de que se modifique la base de datos. Una vez que exista el registro de bitácora podemos sacar la modificación de la base de datos si esto es deseable. También tenemos la posibilidad de deshacer una modificación que ya se ha escrito en la base de datos.

Para que los registros de bitácora sean útiles en la recuperación de fallos del sistema y del disco, el registro debe residir en memoria estable.

Modificación diferida de la base de datos

La técnica de modificación diferida garantiza la atomicidad de la transacción grabando todas las modificaciones de la base de datos en la bitácora, pero aplazando la ejecución de todas las operaciones write de una transacción hasta que la transacción se comete parcialmente.

Cuando una transacción está parcialmente cometida, la información en la bitácora asociada a la transacción se usa en la ejecución de las escrituras diferidas. Si el sistema se cae antes de que la transacción termine su ejecución, o si la transacción aborta, entonces simplemente se ignora la información en la bitácora.

El sistema puede recuperarse de cualquier fallo que no resulte en la pérdida de información de almacenamiento no volátil. Si ocurre algún fallo, el esquema de recuperación recorre la bitácora y se vuelve a hacer cada una de las transacciones cometidas con la operación **redo**.

Redo (T_i) asigna los nuevos valores a todos los datos que actualiza la transacción T_i. La operación redo debe ser idempotente, es decir, ejecutarla varias veces debe ser equivalente a ejecutarla una vez.

Después de ocurrir un fallo, el sistema de recuperación consulta la bitácora para determinar que transacciones necesitan volverse a hacer. La transacción T_i necesita repetirse si, y sólo si, la bitácora contiene tanto el registro <T_i starts> como el registro <T_i commits>.

Modificación inmediata de la base de datos

La técnica de actualización inmediata permite que las modificaciones de la base de datos se graben en la base de datos mientras la transacción está todavía en estado activo. En el caso de que ocurra una caída o un fallo de una transacción el campo del valor antiguo de los registros de bitácora debe utilizarse para restaurar los datos modificados al valor que tenían antes de que comenzara la transacción.

El esquema de recuperación utiliza dos procedimientos de recuperación:

- **undo**(T_i), que restaura todos los datos que T_i actualiza a los valores que tenían anteriormente.
- **redo**(T_i), que asigna los nuevos valores a todos los datos que actualiza la transacción T_i.

Las operaciones undo y redo deben ser idempotentes para garantizar un comportamiento correcto aun cuando ocurra un fallo durante el proceso de recuperación.

Después de ocurrir un fallo, el esquema de transacciones consulta la bitácora para determinar que transacciones necesitan volverse a hacer y cuales necesitan deshacerse.

La transacción T_i debe deshacerse si la bitácora contiene el registro <T_i starts>, pero no contiene el registro <T_i commits>.

La transacción debe volverse a hacer si la bitácora contiene tanto el registro <T_i starts> como el registro <T_i commits>.

Gestión de registros intermedios

Grabar en memoria cada registro de la bitácora insume un gran costo de tiempo. Es conveniente grabar varios registros de bitácora de una vez. Esto significa que un registro de bitácora puede residir en memoria principal un período considerable de tiempo antes de grabarse en memoria estable. Puesto que dichos registros de bitácora se pierden si el sistema se cae, debemos imponer requisitos adicionales en las técnicas de recuperación para garantizar la atomicidad de la transacción.

- La transacción T_i entra en el estado cometido después de haberse grabado en memoria estable el registro de bitácora <T_i commit>.

- Antes de que el registro de bitácora <T_i commit> pueda grabarse en memoria estable, deben haberse grabado en memoria estable todos los registros de bitácora que pertenecen a T_i.
- Antes de grabar en la base de datos un bloque que está en memoria principal, deben haberse grabado en memoria estable todos los registros de bitácora que pertenecen a los datos de ese bloque.

Puntos de verificación

Cuando se utilizan los esquemas de recuperación con bitácora, se recorre toda la bitácora y se lleva a un estado consistente la base de datos por medio de las operaciones undo y redo. Esto trae dos problemas:

- El proceso de búsqueda consume mucho tiempo.
- La mayor parte de las transacciones que necesitan volver a hacerse ya escribieron sus modificaciones en la base de datos, y aunque no hace nada volver a hacerlas, se pierde mucho tiempo.

Para evitar estos problemas se utilizan puntos de verificación. Cuando se ejecuta una transacción, además de mantener la bitácora, el sistema realiza puntos de verificación, ante ellos se debe:

- Guardar los registros de bitácora que estaban en memoria principal a almacenamiento estable.
- Grabar en disco todos los bloques modificados de los registros intermedios (buffer).
- Grabar un registro de la bitácora <checkpoint> en memoria estable.

Ahora, si una transacción se ejecuta antes del checkpoint, el registro <T_i commit> aparece antes del checkpoint, por lo cual la modificación de la base de datos se realizo antes del checkpoint o como parte de este, entonces no es necesario realizar la operación redo de esa transacción.

Ante un fallo, se recorre hacia atrás la bitácora hasta encontrar el primer checkpoint. Luego, se busca el registro <T_i start> siguiente.

Entonces ahora se deben rehacer la transacción T_i y todas las transacciones T_j tales que T_j se haya ejecutado después que T_i, los demás registros de bitácora se ignoran.

Ventaja: cuanto más frecuentes son los checkpoint, más rápido se recupera el sistema.

Desventaja: si no ocurren fallos, se tiene el costo de guardar la información asociada.

Doble paginación

Una alternativa a las técnicas de recuperación de caídas basadas en bitácoras es la doble paginación. Es posible que la doble paginación requiera menos accesos a disco que los métodos de bitácora.

La base de datos se divide en cierto número de bloques de longitud fija, que se denominan **páginas**. No es necesario almacenar estas páginas en un orden determinado en el disco, pero debe haber una forma de encontrar la página i-ésima de la base de datos para cualquier i dado. Esto se logra teniendo una tabla de paginación.

La tabla de paginación tiene n entradas, una por cada página de la base de datos. Cada entrada tiene un puntero a una página del disco.

La idea en que se basa la técnica de paginación es mantener dos tablas de paginación durante la vida de una transacción, la **tabla de paginación actual** y la **tabla de paginación doble**. La tabla de paginación doble no se modifica en ningún momento durante la transacción. La tabla de paginación actual puede cambiarse cuando la transacción realiza una operación write.

Supóngase que las transacciones realizan una operación write(X, x_j) y que X reside en la página i-ésima. La operación write se ejecuta de la siguiente manera:

1) Si la página i-ésima no está todavía en la memoria principal, entonces se ejecuta input(X).
2) Si es la primera escritura que realiza esta transacción en la página i-ésima, entonces se modifica la tabla de paginación actual así:
    a) Se localiza una página que no esté usada en el disco.
    b) Se borra la página hallada en a) de la lista de páginas libres.
    c) Se modifica la tabla de paginación actual de forma que la entrada i-ésima apunte a la página encontrada en a).
3) Se asigna el valor de x_j a X en la página que está en el buffer.

Cuando la transacción termina, la tabla de paginación actual se convierte en la nueva tabla de paginación doble, y se permite que comience a ejecutarse la siguiente transacción. Es importante que la

tabla de paginación doble se almacene en memoria no volátil, ya que es la única forma de localizar las páginas de la base de datos.

Los abortos son automáticos, ya que se tiene la dirección de la página anterior sin modificaciones.

Para cometer una transacción debemos hacer lo siguiente:

1) Asegurarse de que todas las páginas del registro intermedio (buffer) en memoria principal que haya modificado la transacción se graben en disco.
2) Grabar en disco la tabla de paginación actual.
3) Grabar la dirección en disco de la página actual en la posición fija de memoria estable que contiene la dirección de la tabla de paginación doble. Esto se escribe sobre la dirección de la tabla de paginación doble antigua. Por lo tanto, la tabla de paginación actual se convierte en la tabla de paginación doble y la transacción está cometida.

Si ocurre una caída antes de completar el paso 3, volvemos al estado que existía antes de ejecutar la transacción. Si la caída ocurre después de completarse el paso 3, los efectos de la transacción se conservarán.

Ventajas de la paginación doble sobre la bitácora:

- Se elimina el tiempo extra requerido para grabar registros.
- La recuperación de las caídas es bastante más rápida, ya que no se necesitan operaciones undo y redo.

Desventajas de la paginación doble sobre la bitácora:

- Fragmentación de los datos: las páginas de la base de datos cambian de posición cuando se actualizan.
- Recolección de basura: cada vez que se comete una transacción, las páginas de la base de datos que contienen la versión anterior de los datos que la transacción modificó se vuelven inaccesibles. Tales páginas se consideran basura, ya que no son parte del espacio libre y no tienen información útil. También puede crearse basura como efecto secundario de las caídas. Periódicamente es necesario localizar estas páginas y añadirlas a la lista de páginas libres.
- Es difícil de adaptar para sistemas que permiten que se ejecuten varias transacciones concurrentemente.

Fallo con pérdida de memoria no volátil

Para recuperarse de un fallo que resulta en la pérdida de información no volátil es necesario volcar todo el contenido de la base de datos a almacenamiento estable. Si ocurre un fallo que resulta en la perdida de bloques físicos, se utiliza el último volcado para llevar al sistema a un estado consistente anterior. Una vez hecho esto, se usa la bitácora para llevar al sistema de base de datos a un estado consistente más reciente.

Ninguna transacción debe estar activa durante el volcado y debe llevarse a cabo un procedimiento similar al de puntos de verificación:

1) Grabar en memoria estable todos los registros de la bitácora que residen actualmente en memoria principal.
2) Grabar todos los bloques de registro intermedio (buffer) en el disco.
3) Copiar los contenidos de la base de datos en memoria estable.
4) Grabar un registro de bitácora <**dump**> en memoria estable.

El procedimiento de volcado es costoso porque toda la base de datos debe copiarse en memoria estable que requiere una transferencia de datos considerable y dado que las transacciones se paran, se pierden ciclos de CPU.

Implementación de memoria estable

La información que reside en memoria estable nunca se pierde.

La transferencia de bloques entre la memoria y el disco puede resultar en:

- Terminación con éxito.
- Fallo parcial: ocurrió un fallo durante la transferencia y el bloque de destino tiene información incorrecta.
- Fallo total: ocurrió un fallo bastante pronto durante la transferencia, de forma que el bloque destino permanece intacto.

El sistema debe mantener dos bloques físicos por cada bloque lógico de la base de datos. Una operación de salida se ejecuta como sigue:

1) Escribir la información en el primer bloque físico.
2) Una vez que se completa con éxito la primera escritura, escribir la misma información en el segundo bloque físico.
3) La salida está completa sólo después de terminar con éxito la segunda escritura.

Durante la recuperación se examina cada par de bloques físicos. Si los dos son iguales y no existen errores detectables, no es necesario tomar más acciones. Si un bloque contiene un error detectable, se sustituye su contenido por el valor del segundo bloque. Si ambos bloques contienen errores no detectables, pero de contenido distinto, entonces se sustituye el contenido del primer bloque por el valor del segundo. Este procedimiento de recuperación garantiza que una escritura en almacenamiento estable termine con éxito o no produzca cambio alguno. El intento de escribir en almacenamiento estable tiene éxito solo si se escriben todas las copias.

Violaciones de la seguridad e integridad

El mal uso que se haga de la base de datos puede ser intencionado o accidental. La pérdida accidental de la consistencia de los datos puede deberse a:

- Caídas durante el procesamiento de las transacciones.
- Anomalías por acceso concurrente a la base de datos.
- Anomalías que resultan de la distribución de los datos entre varias computadoras.
- Un error lógico que viola la suposición de que las transacciones respetan las protecciones de consistencia de la base de datos.

Algunas de las formas de acceso indebido son:

- Lectura de datos sin autorización.
- Modificación de datos sin autorización.
- Destrucción no autorizada de los datos.

El término **seguridad** de la base de datos normalmente se refiere a la protección contra el acceso mal intencionado, mientras que **integridad** se refiere a la protección contra una pérdida accidental de consistencia.

Para proteger la base de datos es necesario adoptar medidas de seguridad en varios niveles:

- **Físico**: hay que proteger a las localidades físicamente contra la penetración clandestina de intrusos.
- **Humano**: se debe tener cuidado al autorizar a usuarios que permitan el acceso de intrusos.
- **Sistema operativo**: hay que tener cuidado ya que el sistema operativo puede servir a que otros usuarios accedan a la base de datos sin autorización.
- **Sistemas de base de datos**: debe garantizar que todos los usuarios que accedan cumplan con los requisitos de acceso.

La seguridad en todos los niveles anteriores debe mantenerse para asegurar la seguridad de la base de datos. Un punto débil en un nivel bajo de seguridad permite que se burlen medidas estrictas de seguridad a alto nivel.

El control de integridad se logra exigiendo que los datos almacenados en la base de datos cumplan con ciertos requisitos de integridad, con el fin de prevenir pérdida accidental de la consistencia.

Autorizaciones y vistas

Una vista puede ocultar datos que el usuario no tiene necesidad de ver. Esta posibilidad sirve tanto para simplificar la utilización del sistema como para fomentar la seguridad. La seguridad se logra si se cuenta con un mecanismo que limite a los usuarios a su vista o vistas personales. Lo normal es que las bases de datos relacionales cuenten con dos niveles de seguridad:

- Relación: puede permitirse o impedirse que un usuario tenga acceso directo a una relación.
- Vista: puede permitirse o impedirse que el usuario tenga acceso a la información que aparece en una vista.

Es posible utilizar una combinación de seguridad al nivel relacional y al nivel de vistas para limitar el acceso del usuario exclusivamente a los datos que necesita.

Un usuario puede tener varias formas de autorización sobre partes de la base datos y sobre el esquema de la base de datos. Por ejemplo: autorización de lectura, de inserción, de actualización y de

borrado de la base de datos; autorización de índice (permite la creación y eliminación de índices), autorización de recursos (permite la creación de relaciones nuevas), autorización de alteración (permite agregar o eliminar atributos de una relación) y autorización de eliminación (permite eliminar relaciones).

Cifrado

Es posible que todas las precauciones que tome la base de datos para evitar el acceso sin autorización no sean suficientes para proteger los datos muy importantes. En estos casos, la información puede cifrarse. No es posible leer datos cifrados a menos que el lector sepa como descifrar la información.

Las buenas técnicas de cifrado tienen las siguientes propiedades:

- Para los usuarios autorizados es relativamente sencillo cifrar y descifrar los datos.
- El esquema de cifrado no depende de mantener en secreto el algoritmo, sino de un parámetro del algoritmo llamado clave de cifrado.
- Para un intruso es muy difícil determinar cual es la clave de cifrado.

Para que este sistema funcione es necesario proporcionar claves de cifrado autorizados a través de un mecanismo seguro. Esto es un punto débil, ya que la seguridad del sistema dependerá de la seguridad del mecanismo de transmisión de la clave de cifrado.

El cifrado de clave pública, se basa en dos claves, una clave pública y una clave privada. Cada uno de los usuarios U_i tiene sus propias claves, pública E_i y privada D_i. Todas las claves públicas se publican, pero la privada la conoce sólo el usuario al que pertenece.

Si el usuario U_1 desea almacenar datos cifrados, los codificará empleando su clave pública E_1; para descifrarla es necesario contar con la clave privada D_1.

Dado que la clave de cifrado de cada usuario es pública, es posible intercambiar información de forma segura utilizando esta técnica. Si el usuario U_1 quiere compartir sus datos con U_2, U_1 codificará los datos empleando la clave pública de U_2 que es E_2. Como el único que sabe como descifrar los datos es el usuario U_2, la transferencia de información puede realizarse con seguridad.

Unidad 7

Bases de datos en sistemas concurrentes

Los beneficios de la multiprogramación son un mejor aprovechamiento del procesador y una productividad total de transacciones más alta, es decir, la cantidad de trabajo que se realiza en un intervalo de tiempo dado. Además, en el caso de transacciones interactivas en las que el usuario espera el resultado, el tiempo de respuesta debe ser lo más corto posible.

Planificación

Cuando se ejecutan varias transacciones de manera concurrente, la consistencia de la base de datos puede destruirse aun cuando cada una de las transacciones individuales sea correcta.

Una **planificación** representa el orden cronológico en que se ejecutan las instrucciones en el sistema. Una planificación para un conjunto de transacciones debe constar de todas las instrucciones de esas transacciones y debe conservar el orden en que aparecen las instrucciones en cada transacción individual.

Una **planificación en serie** consta de una secuencia de instrucciones de varias transacciones en las que las instrucciones pertenecientes a una transacción aparecen juntas en esa planificación. Así, para un conjunto de n transacciones existen n! planificaciones en serie diferentes.

Cuando se ejecutan varias transacciones concurrentemente, no es necesario que la planificación correspondiente este en serie. Así, el número de planificaciones posibles para un conjunto de n transacciones es mucho mayor que n!.

Es necesario que una transacción sea un programa que conserve la consistencia. Es decir, cada transacción, al ejecutarse sola, transfiere el sistema de un estado consistente a un nuevo estado consistente. Sin embargo, durante la ejecución de una transacción puede que el sistema entre temporalmente en un estado inconsistente. La inconsistencia temporal es la que causa la inconsistencia en las **planificaciones en paralelo**.

Una planificación después de su ejecución debe dejar la base de datos en un estado consistente. Además, la planificación debe ser, de alguna forma, equivalente a una planificación en serie.

Las únicas operaciones significativas de una transacción, desde el punto de vista de una planificación, son las instrucciones de read y write.

Conflicto en planificaciones serializables

Sea una planificación S en la que hay dos instrucciones consecutivas I_i e I_j de las transacciones T_i y T_j, respectivamente. Si I_i e I_j se refieren a diferentes datos, entonces podemos intercambiar I_i e I_j sin que afecte a los resultados de las instrucciones de la planificación. Sin embargo, si I_i e I_j se refieren al mismo dato Q, entonces puede que importe el orden de los dos pasos. Se debe considerar:

- $I_i = I_j$ = read (Q): el orden no importa, ya que leen el mismo valor sin tener en cuenta el orden.
- I_i = read (Q) e I_j = write (Q): si I_i viene antes que I_j, entonces T_i no lee el valor de Q que escribe T_j en la instrucción I_j. Si I_j viene antes que I_i, entonces T_i lee el valor de Q que escribe T_j. Por tanto, el orden de I_i e I_j sí importa.
- I_i = write (Q) e I_j = read (Q): igual al caso anterior.
- $I_i = I_j$ = write (Q): el orden de estas instrucciones no afecta a T_i ni a T_j. Sin embargo afecta al valor obtenido por la siguiente instrucción read de S, ya que en la base de datos solamente se conserva el resultado de la última de las dos instrucciones write.

Decimos que I_i e I_j están en **conflicto** si son operaciones de transacciones distintas sobre el mismo dato, y por lo menos una de estas instrucciones es una operación write.

Sean I_i e I_j instrucciones consecutivas de una planificación S. Si I_i e I_j son instrucciones de diferentes transacciones e I_i e I_j no están en conflicto, entonces podemos intercambiar el orden de I_i e I_j para producir una nueva planificación S'.

Si una planificación S puede transformarse en una planificación S' mediante una serie de intercambios de instrucciones no conflictivas, decimos que S y S' son **equivalentes en cuanto a conflictos**.

Decimos que una planificación S es serializable en cuanto a conflictos si es equivalente en conflictos a una planificación en serie.

Serializabilidad de vistas

Sean dos planificaciones S y S' en las que el mismo conjunto de transacciones participa en ambas planificaciones. Se dice que las planificaciones S y S' son **equivalentes en cuanto a vistas** si se cumplen las tres condiciones siguientes:

1) Para cada dato Q, si la transacción T_i lee el valor inicial de Q en la planificación S, entonces la transacción T_i también debe leer el valor inicial de Q en la planificación S'.
2) Para cada dato Q, si la transacción T_i ejecuta read(A) en la planificación S, y ese valor fue producido por la transacción T_j (si existe), entonces la transacción T_i también debe leer en la planificación S' el valor de Q que fue producido por la transacción T_j.
3) Para cada dato Q, la transacción (si existe) que ejecuta la operación write(Q) final en la planificación S debe ejecutar la operación final write(Q) en la planificación S'.

Las condiciones 1 y 2 aseguran que cada transacción lee los mismos valores en ambas planificaciones y, por lo tanto, realiza el mismo cálculo. La condición 3, junto a la 1 y 2, asegura que ambas planificaciones resultan en el mismo estado final del sistema.

Decimos que una planificación S es **serializable en vistas** si es equivalente en vistas a una planificación en serie.

A las operaciones que realizan una operación write(Q) sin haber realizado una operación read(Q) se las denomina **escrituras ciegas**. Las escrituras ciegas aparecen en cualquier planificación serializable en vistas que no sea serializable en conflictos.

Pruebas de serializabilidad de conflictos

Sea S una planificación. Construimos un grafo dirigido llamado grafo de precedencia de S. Este grafo consta de un par G = (V, E) donde V es un conjunto de vértices y E es un conjunto de aristas. El conjunto de vértices consta de todas las transacciones que participan en la planificación. El conjunto de aristas consta de todas las aristas $T_i \rightarrow T_j$ para las cuales se cumple una de las tres condiciones siguientes:

- T_i ejecuta write(Q) antes de que T_j ejecute read(Q).
- T_i ejecuta read(Q) antes de que T_j ejecute write(Q).
- T_i ejecuta write(Q) antes de que T_j ejecute write(Q).

Si existe una arista $T_i \rightarrow T_j$ en el grafo de precedencia, esto implica que en cualquier planificación en serie S' equivalente a S, T_i debe aparecer antes de T_j.

Si el grafo de precedencia de S tiene un ciclo, entonces la planificación S no es serializable en conflictos. Si el grafo no contiene ciclos, entonces la planificación S es serializable en conflictos. El orden de serializabilidad puede obtenerse mediante una ordenación topológica, que determina un orden lineal consistente con el orden parcial del grafo de precedencia.

Así, para probar si existe serializabilidad en conflictos, necesitamos construir el grafo de precedencia y aplicar un algoritmo de detección de ciclos. Puesto que encontrar un ciclo en un grafo requiere del orden de n^2 operaciones, donde n es el número de vértices del grafo, tenemos un esquema práctico para determinar la serializabilidad en conflictos.

Bloqueo de datos

Una forma de asegurar la serializabilidad es exigir que el acceso a los datos se haga de manera mutuamente excluyente; es decir, que mientras una transacción accede a un dato, ninguna otra puede modificarlo. El método más común que se utiliza para implementar esto es permitir que una transacción acceda a un dato solo si tiene actualmente un bloqueo en ese dato.

Hay dos tipos de bloqueo:

- Compartido (S): si una transacción T_i ha obtenido un bloqueo de modo compartido en el dato Q, entonces T_i puede leer este dato pero no puede escribir Q.
- Exclusivo (X): si una transacción T_i ha obtenido un bloqueo de modo exclusivo en el dato Q, entonces T_i puede leer y escribir Q.

Todas las transacciones deben pedir un bloqueo en el modo adecuado en el dato Q dependiendo del tipo de operaciones que van a realizar sobre Q.

Sean A y B modos de bloqueo arbitrarios. Supóngase que una transacción T_i solicita un bloqueo de modo A en el dato Q en el cual la transacción T_j (T_i distinta de T_j) tiene actualmente un bloqueo de modo B.

Si se le puede conceder a la transacción T_i un bloqueo en Q inmediatamente, a pesar de la presencia del bloqueo de modo B, entonces decimos que el modo A es compatible con el modo B.

Cuando se llega a un estado en el que ninguna transacción puede seguir su ejecución normal, esta situación se denomina **bloqueo**. Cuando ocurre un bloqueo, el sistema debe retroceder una de las transacciones.

Todas las transacciones deben seguir un conjunto de reglas llamado protocolo de bloqueo, que indica cuando una transacción puede bloquear y desbloquear cada uno de sus datos. Los **protocolos de bloqueo** restringen el número de planificaciones en un subconjunto propio de todas las planificaciones serializables posibles.

Decimos que una planificación S es **legal** bajo un protocolo de bloqueo dado si S es una planificación posible para un conjunto de transacciones que sigue las reglas del protocolo de bloqueo. Decimos que un protocolo de bloqueo **garantiza la serializabilidad** en conflictos solo si para todas las planificaciones legales la relación asociada es acíclica.

Protocolo de bloqueo basado en dos fases

Este protocolo requiere que todas las transacciones hagan sus solicitudes de bloqueo y desbloqueo en dos fases:

- Fase de crecimiento: una transacción puede obtener bloqueos pero no puede liberar ningún bloqueo.
- Fase de encogimiento: una transacción puede liberar bloqueos pero no puede obtener ningún bloqueo nuevo.

Inicialmente, una transacción está en la fase de crecimiento. La transacción adquiere los bloqueos que necesita. Una vez que la transacción libere un bloqueo, entra en la fase de encogimiento y no podrá solicitar más bloqueos.

Este protocolo garantiza serializabilidad en conflictos pero no garantiza que no se presenten situaciones de bloqueos.

Si T_i no es una transacción de dos fases, siempre es posible encontrar otra transacción T_j que sea de dos fases, tal que exista una posible planificación no serializable en conflictos pata T_i y T_j.

La conversión de bloqueos no puede permitirse que ocurra de manera arbitraria. La elevación (compartido a exclusivo) sólo puede tener lugar en la fase de crecimiento, mientras que la conversión de modo exclusivo a compartido sólo puede tener lugar en la fase de encogimiento.

Protocolos basados en grafos

Requiere que tengamos conocimiento previo del orden en que se va a acceder a los datos de la base de datos. Dada esa información, es posible construir protocolos de bloqueo que no sean de dos fases pero que de todos modos garanticen la serializabilidad.

Para adquirir ese conocimiento previo imponemos una ordenación parcial → en el conjunto $D = \{d_1, d_2, .., d_n\}$ de todos los datos. Si $d_i \rightarrow d_j$, entonces cualquier transacción que acceda tanto a d_i como a d_j debe acceder a d_i antes de acceder a d_j.

La ordenación parcial implica que el conjunto D puede verse como un grafo acíclico dirigido, llamado grafo de base de datos. Se bloquean los datos en modo exclusivo. Este protocolo es llamado **protocolo de árbol**.

En el protocolo de árbol la única instrucción de bloqueo permitida es lock-X. Cada transacción T_i puede bloquear un dato a lo sumo una vez y debe respetar las siguientes reglas:

1) El primer bloqueo de T_i puede ser en cualquier dato.
2) A partir de ese momento, T_i puede bloquear un dato Q solo si T_i bloquea actualmente al padre de Q.
3) Los datos pueden desbloquearse en cualquier momento.
4) Un dato que T_i haya bloqueado y desbloqueado, T_i no puede volver a bloquearlo posteriormente.

Todas las planificaciones que sean legales bajo el protocolo de árbol son serializables en conflictos.

El protocolo de árbol garantiza no sólo la serializabilidad, sino también la libertad de desbloqueo.

El protocolo de bloqueo de árbol tiene la ventaja con respecto al protocolo de bloqueo de dos fases de que el desbloqueo puede ocurrir antes. Esto puede resultar en menos tiempo de espera y en un aumento de la concurrencia. Además, puesto que hay libertad de desbloqueo, no se requieren retrocesos. Sin embargo, tiene la desventaja de que en algunos casos es necesario que una transacción bloquee datos a los que no accede.

Protocolos basados en hora de entrada

Para determinar la serializabilidad se determina un orden entre las transacciones por adelantado.

A cada transacción T_i del sistema le asociamos una hora de entrada fija única, representada por $TS(T_i)$. El sistema de base de datos asigna esta hora de entrada antes de que la transacción T_i empiece su ejecución. Si se asignó la hora de entrada $TS(T_i)$ a la transacción T_i y entra una nueva transacción T_j al sistema, entonces $TS(T_i) < TS(T_j)$. Como hora de entrada se puede utilizar el valor del reloj del sistema o un contador lógico que se incremente después de asignar una nueva hora de entrada.

Las horas de entrada de las transacciones determinan el orden de serializabilidad.

Se asocian dos valores de hora de entrada a cada dato:

- **W-hora-de-entrada**(Q), que representa la mayor hora de entrada de cualquier transacción que ejecutó con éxito write(Q).
- **R-hora-de-entrada**(Q), que representa la mayor hora de entrada de cualquier transacción que ejecutó con éxito read(Q).

Estas horas de entrada se actualizan cada vez que se ejecuta una nueva instrucción read(Q) o write(Q).

El protocolo de hora de entrada garantiza que todas las operaciones read y write que pudieran entrar en conflicto se ejecuten en el orden de hora de entrada.

Algoritmo de ejecución

- T_i solicita read(Q):
  - ✓ Si $TS(T_i) <$ W-hora-de-entrada(Q), entonces T_i necesita leer un valor de Q que ya fue sobrescrito. Por lo tanto, la operación read se rechaza y T_i retrocede.
  - ✓ Si $TS(T_i) \geq$ W-hora-de-entrada(Q), entonces la operación read se ejecuta, y R-hora-de-entrada(Q) se pone al valor máximo de R-hora-de-entrada(Q) y $TS(T_i)$.
- T_i solicita write(Q):
  - ✓ Si $TS(T_i) <$ R-hora-de-entrada(Q), entonces el valor de Q que T_i está produciendo se necesito con anterioridad y se supuso que nunca se produciría. Por lo tanto, la operación write se rechaza y T_i retrocede.
  - ✓ Si $TS(T_i) <$ W-hora-de-entrada(Q), entonces T_i está intentando escribir un valor obsoleto de Q. Por lo tanto, esta operación write se rechaza y T_i retrocede.
  - ✓ En los demás casos, la operación write se ejecuta, y W-hora-de-entrada(Q) se pone al valor máximo de W-hora-de-entrad(Q) y $TS(T_i)$.

A una transacción que retrocede el esquema de control de concurrencia como resultado de solicitar una operación read o write, se le asigna una nueva hora de entrada y se reinicia.

Operaciones insertar y suprimir

Si se emplea bloqueo en dos fases, se requiere un bloqueo exclusivo en un dato antes de que pueda **eliminarse**. Bajo el protocolo de ordenación por hora de entrada se debe realizar una prueba similar a la de write.

- Si $TS(T_i) <$ R-hora-de-entrada(Q), entonces el valor de Q que T_i iba a suprimir ya ha sido suprimido por una transacción T_j con $TS(T_j) > TS(T_i)$. Por lo tanto, la operación delete se rechaza y T_i retrocede.
- Si $TS(T_i) <$ W-hora-de-entrada(Q), entonces una transacción T_j con $TS(T_j) > TS(T_i)$ ha escrito Q. Por lo tanto, esta operación delete se rechaza y T_i retrocede.
- En caso contrario, la operación delete se ejecuta.

Dado que un insert(Q) asigna un valor al dato Q, un **insert** se trata de la misma manera que un write para propósitos de control de concurrencia:

- Bajo el protocolo de bloqueo de dos fases, si T_i realiza una operación insert(Q), a T_i se le da un bloqueo exclusivo en el dato Q recién creado.
- Bajo el protocolo de ordenación por hora de entrada, si T_i realiza una operación insert(Q), los valores R-hora-de-entrada(Q) y W-hora-de-entrada(Q) se ponen a $TS(T_i)$.

Retroceso en cascada

Para recuperarse correctamente del fallo de una transacción T, puede ser necesario retroceder varias transacciones. Estas situaciones ocurren si las transacciones han leído datos que T ha escrito.

Este fenómeno, en el que un simple fallo de una transacción conduce a una serie de retrocesos de transacciones, se llama **retroceso en cascada**.

El retroceso en cascada puede ocurrir bajo el bloqueo en dos fases.

El protocolo basado en hora de entrada puede dar como resultado el retroceso en cascada.

El retroceso en cascada no es deseable, ya que lleva a deshacer una importante cantidad de trabajo.

Planificaciones recuperables

Los sistemas de procesamiento de transacciones deben garantizar que es posible recuperarse del fallo de cualquier transacción activa. Así pues, deben evitarse las situaciones no recuperables.

El protocolo de **hora de entrada** puede modificarse para prevenir ejecuciones no recuperables y evitar los retrocesos en cascada como sigue: se asocia un **bit de ejecución** b_i a cada transacción T_i. Inicialmente b_i es falso. Cuando T_i se ejecuta, b_i se pone a verdadero. Una transacción T_i que intenta leer un dato Q debe satisfacer los requisitos del protocolo de hora de entrada. Si lo hace, se comprueba el bit de ejecución de la transacción que fue el último en escribir Q. Si este bit es verdadero, se permite que T_j haga la lectura. En caso contrario, T_j debe esperar a que el bit se ponga a verdadero.

Esta modificación del protocolo de hora de entrada introduce esperas, pero no es posible el bloqueo.

Bajo el **bloqueo de dos fases**, el retroceso en cascada puede evitarse imponiendo el requisito adicional de que todos los bloqueos exclusivos que se ha puesto una transacción T deben conservarse hasta que T termine. Esto garantiza que cualquier dato escrito por una transacción que no se ha ejecutado está bloqueado en modo exclusivo, previniendo que cualquier otra transacción lea los datos.

Exploración de bitácora

En un sistema de procesamiento de transacciones concurrentes es necesario que el registro de bitácora de puntos de verificación sea de la forma <**checkpoint** L>, donde L es una lista de transacciones activas en el momento del punto de verificación. Cuando el sistema se recupera de una caída construye dos listas: la **lista de deshacer**, que consta de las transacciones a deshacer, y la **lista de volver a hacer**, que consta de las transacciones que se deben repetir.

Estas dos listas inicialmente están vacías. Examinamos la bitácora hacia atrás, registro por registro, hasta que se encuentre el primer registro <checkpoint>:

- Por cada registro que se encuentre de la forma <T_i commits>, añadimos T_i a la lista de volver a hacer.
- Por cada registro que se encuentre de la forma <T_i starts>, si T_i no está en la lista de volver a hacer añadimos T_i a la lista de deshacer.

Cuando se han examinado todos los registros de la bitácora apropiados, comprobamos la lista L. Para cada transacción T_i en L, si T_i no está en la lista de volver a hacer entonces añadimos T_i a la lista de deshacer.

Una vez que se han construido las dos listas, la recuperación procede así:

1) Se vuelve a examinar hacia atrás la bitácora del registro más reciente y se realiza undo(T_i) para cada T_i en la lista de deshacer.
2) Se continúa la exploración de la bitácora hacia atrás hasta que se haya localizado el registro <T_i starts> para todas las T_i en la lista de volver a hacer.
3) Se examina la bitácora hacia delante y se realiza redo(T_i) para cada T_i en la lista de volver a hacer.

Gestión de bloqueos

Un sistema está en un estado de bloqueo si existe un conjunto de transacciones en espera $\{T_0,T_1,..,T_n\}$ tal que T_0 está esperando un dato que tiene T_1, y T_1 está esperando un dato que tiene T_2, y... y T_{n-1} está esperando un dato que tiene T_n y T_n está esperando un dato que tiene T_0.

Existen dos métodos principales para tratar el problema del bloqueo:

- Usar un protocolo de prevención de bloqueo para garantizar que el sistema nunca entrará en un estado de bloqueo.
- Permitir que el sistema entre en un estado de bloqueo y entonces intentar la recuperación utilizando un esquema de detección y recuperación de bloqueo.

Ambos métodos pueden resultar en retroceso de transacciones.

Prevención de bloqueos

El esquema más sencillo requiere que cada transacción bloquee todos sus datos antes de comenzar a ejecutarse. Este protocolo tiene dos desventajas importantes: la utilización de los datos puede ser muy baja, ya que muchos datos pueden estar bloqueados aunque no se usen durante un largo período de tiempo, y además existe la posibilidad de inanición.

Otro método es imponer una ordenación parcial de todos los datos y exigir que una transacción pueda bloquear un dato solo en el orden especificado por el orden parcial.

Otro enfoque es utilizar la expropiación y el retroceso de transacciones. Para controlar la expropiación, se asigna una hora de entrada única a cada transacción. Estas horas de entrada se usan para decidir si una transacción debe esperar o retroceder. Si una transacción retrocede, conservará su hora de entrada antigua cuando se reinicie. Hay dos esquemas:

- **Esperar - morir**: se basa en una técnica que no utiliza expropiaciones. Cuando la transacción T_i solicita un dato que T_j tiene en ese momento, sólo se permitirá que T_i espere si tiene una hora de entrada menor que la de T_j. En caso contrario T_i retrocede (muere).
- **Herir - esperar**: se basa en una técnica de expropiaciones y es una parte que contrarresta al esquema esperar - morir. Cuando la transacción T_i solicita un dato que T_j tiene en ese momento, sólo se permitirá que T_i espere si tiene una hora de entrada mayor que la de T_j. En caso contrario, T_j retrocede (se hiere).

Ambos esquemas evitan la inanición. Esto se debe a que en un momento dado siempre existe una transacción que tiene la hora de entrada más baja.

El problema principal de estos esquemas es que pueden ocurrir retrocesos innecesarios.

Detección y recuperación de bloqueos

Los bloqueos pueden describirse de manera precisa en términos de un grafo dirigido denominado grafo de espera. Este grafo consta de un par de vértices G = (V, E), donde V es un conjunto de vértices y E es un conjunto de aristas. El conjunto de vértices consta de todas las transacciones del sistema. Cada elemento del conjunto E de aristas es un par ordenado $T_i \rightarrow T_j$. Si $T_i \rightarrow T_j$ está en E, entonces existe una arista dirigida de la transacción T_i a la T_j, lo que implica que la transacción T_i está esperando a que la transacción T_j libere un dato que ella necesita.

Cuando la transacción T_i solicita un dato que tiene en ese momento la transacción T_j, se inserta la arista $T_i \rightarrow T_j$ en el grafo de espera.

Para detectar los bloqueos, el sistema necesita mantener el grafo de espera e invocar periódicamente un algoritmo que busque un ciclo en el grafo.

Cuando un algoritmo de detección determina que existe un bloqueo, el sistema debe recuperarse del bloqueo. La solución más común es retroceder una o más transacciones para romper el bloqueo. Para hacerlo se necesita tener en cuenta tres factores:

- Selección de una víctima: se debe determinar que transacción o transacciones retroceder para romper el bloqueo. Se deben retroceder aquellas transacciones que tengan un costo mínimo.
- Retroceso: se debe determinar hasta donde se debe retroceder. Es más efectivo retroceder la transacción solo lo necesario hasta romper el bloqueo. Sin embargo, este método requiere que el sistema mantenga información adicional referente al estado de todas las transacciones activas.
- Inanición: debemos asegurarnos que una transacción puede elegirse como víctima sólo un número finito (pequeño) de veces. La solución más común es incluir el número de retrocesos en el factor de costo.

Unidad 8

Bases de datos distribuidas

La diferencia principal entre los sistemas de base de datos centralizados y distribuidos es que, en los primeros, los datos residen en una sola localidad, mientras que, en los últimos, se encuentran en varias localidades.

Un sistema distribuido de base de datos consiste en un conjunto de localidades, cada una de las cuales mantiene un sistema de base de datos local. Cada localidad puede procesar **transacciones locales**, es decir, aquellas que sólo acceden a datos que residen en esa localidad. Además, una localidad puede participar en la ejecución de **transacciones globales**, es decir, aquellas que acceden a datos de varias localidades. La ejecución de transacciones globales requiere comunicación entre las localidades.

Consideraciones al distribuir la base de datos

Ventajas de la distribución de datos

La principal ventaja de los sistemas distribuidos de base de datos es la capacidad de compartir y acceder a la información de una forma fiable y eficaz.

- Utilización compartida de los datos y distribución del control: si varias localidades diferentes están conectadas entre sí, entonces un usuario de una localidad puede acceder a datos disponibles en otra localidad. La ventaja principal de compartir los datos por medio de la distribución es que cada localidad pueda controlar hasta cierto punto los datos almacenados localmente.
- Fiabilidad y disponibilidad: si se produce un fallo en una localidad en un sistema distribuido, es posible que las demás localidades puedan seguir trabajando. En particular, si los datos se repiten en varias localidades, una transacción que requiere un dato específico puede encontrarlo en más de una localidad. Así, el fallo de una localidad no implica necesariamente la desactivación del sistema.

  La disponibilidad es fundamental para los sistemas de base de datos que se utilizan en aplicaciones de tiempo real.
- Agilización del procesamiento de consultas: si una consulta comprende datos de varias localidades, puede ser posible dividir la consulta en varias subconsultas que se ejecuten en paralelo en distintas localidades.

Desventajas de la distribución de datos

La desventaja principal de los sistemas distribuidos de base de datos es la mayor complejidad que se requiere para garantizar una coordinación adecuada entre localidades.

El aumento de la complejidad se refleja en:

- Coste de desarrollo de software.
- Mayor posibilidad de errores.
- Mayor tiempo extra de procesamiento: el intercambio de mensajes y los cálculos adicionales que se requieren para coordinar las localidades son una forma de tiempo extra que no existe en los sistemas centralizados.

Diseño de bases de datos distribuidas

Sea una relación r que se va a almacenar en la base de datos. Hay varios factores que deben tomarse en cuenta al almacenar esta relación en la base de datos distribuida. Tres de ellos son:

- Repetición: el sistema mantiene varias copias idénticas de la relación.
- Fragmentación: la relación se divide en varios fragmentos. Cada fragmento se almacena en una localidad diferente.
- Repetición y fragmentación: la relación se divide en varios fragmentos. El sistema mantiene varias copias idénticas de cada uno de los fragmentos.

Repetición de los datos

Si la relación r está repetida, se almacena una copia en dos o más localidades. La repetición tiene varias ventajas y desventajas:

- Disponibilidad: si falla una de las localidades que contienen la relación r, puede disponerse de ésta en otra localidad.
- Mayor paralelismo: en el caso de que la mayor parte de los accesos a una relación r resulten sólo en la lectura de la relación, varias localidades podrán procesar consultas que involucren a r en paralelo. Mientras más copias de r existan, mayor será la probabilidad de que los datos requeridos se encuentren en la localidad donde se está ejecutando la transacción.
- Mayor tiempo extra para actualizaciones: el sistema debe asegurarse de que todas las copias de la relación r sean consistentes. Esto implica que cada vez que se actualice r, la actualización debe propagarse a todas las localidades que contengan copias.

Fragmentación de los datos

Si la relación r está fragmentada, r se dividirá en varios fragmentos r_1, r_2, .., r_n. Estos fragmentos contienen información suficiente para reconstruir la relación r original. Existen dos esquemas diferentes para fragmentar una relación: **fragmentación horizontal** y **fragmentación vertical**.

Fragmentación horizontal

La relación r se divide en varios subconjuntos r_1, r_2, .., r_n. Cada tupla de la relación r debe pertenecer a alguno de los fragmentos de manera que si es preciso pueda reconstruirse la relación original.

Un fragmento puede definirse como una selección en la relación global r. Es decir, se utiliza un predicado P_i para construir el fragmento r_i de la siguiente manera: $r_i = \sigma_{P1}(r)$.

La reconstrucción de la relación r puede obtenerse al calcular la unión de todos los fragmentos.

Fragmentación vertical

La fragmentación vertical de r(R) involucra la definición de varios subconjuntos, R_1, R_2, .., R_n, de R tales que: $R = R_1 \cup R_2 \cup ... \cup R_n$. Cada fragmento r_i de r se define por: $r_i = \pi_{Ri}(r)$. La relación r puede reconstruirse a partir de los fragmentos realizando el producto natural: $r = r_1 |x| r_2 |x| .. |x| r_n$.

De manera más general, la fragmentación vertical se lleva a cabo añadiendo un atributo especial, llamado id-tupla, al esquema R. Un id-tupla es una dirección física lógica de una tupla. Puesto que cada tupla en r debe tener una dirección única, el atributo id-tupla es una clave del esquema ampliado.

Aunque el atributo id-tupla facilita la implementación de la fragmentación vertical, también es importante que los usuarios no puedan ver este atributo. Si los usuarios tienen acceso a las id-tuplas, el sistema no podrá cambiar las direcciones de las tuplas. Además, el acceso a direcciones internas viola el concepto de independencia de los datos, que es una de las virtudes principales del modelo relacional.

Fragmentación mixta

La relación r se divide en varios fragmentos que constituyen las relaciones r_1, r_2, .., r_n. Cada fragmento se obtiene por la aplicación del esquema de fragmentación, ya sea horizontal o vertical, sobre la relación r, o sobre un fragmento de r que se haya obtenido con anterioridad.

Repetición y fragmentación

La técnica que se acaba de describir para repetir y fragmentar la operación puede aplicarse de manera sucesiva a la misma relación. Es decir, un fragmento puede repetirse, las copias pueden fragmentarse, etc.

Transparencia y autonomía

La **transparencia de la red** es el grado hasta el cual los usuarios del sistema pueden ignorar los detalles del diseño distribuido. La **autonomía local** es el grado hasta el cual el diseñador o administrador de una localidad pueden ser independientes del resto del sistema distribuido.

Asignación de nombres y autonomía local

En una base de datos distribuida, las distintas localidades deben asegurarse no utilizar el mismo nombre para dos datos diferentes.

Una solución para este problema es requerir que se registren todos los nombres en un asignador central de nombres. Sin embargo, este enfoque tiene varias desventajas:

- Es posible que el asignador de nombres se convierta en un cuello de botella.
- Si el asignador de nombres se cae, es posible que ninguna de las localidades del sistema distribuido pueda seguir trabajando.
- Se reduce la autonomía local, ya que la asignación de nombres se controla de forma centralizada.

Un enfoque diferente que origina una mayor autonomía local es exigir que cada localidad ponga como prefijo un identificador de localidad a cualquier nombre que genere. Esto garantiza que dos localidades nunca generarán el mismo nombre. Además, no se requiere un control central.

Esta solución al problema de asignación de nombres logra autonomía local, pero no transparencia de la red, ya que se agregan identificadores de localidad a los nombres.

Cada copia y fragmento de un elemento de información deben tener un nombre único. Es importante que el sistema pueda determinar que copias son copias del mismo elemento de información y que fragmentos son fragmentos del mismo elemento de información.

Transparencia de la repetición y la fragmentación

No es conveniente requerir que los usuarios hagan referencia a una copia especifica de un elemento de información. El sistema debe ser el que determine a que copia debe acceder cuando se le solicite su lectura, y debe modificar todas las copias cuando se produzca una petición de escritura.

Cuando se solicita un dato, no es necesario especificar la copia. El sistema utiliza una tabla - catálogo para determinar cuales son todas las copias de ese dato.

De manera similar, no debe exigirse a los usuarios que sepan como está fragmentado un elemento de información. Un sistema de base de datos distribuido debe permitir las consultas que se hagan en términos de elementos de información sin fragmentar. Esto no presenta problemas graves, ya que siempre es posible reconstruir el elemento de información original a partir de sus fragmentos. Sin embargo, este proceso puede ser ineficiente.

Transparencia de la localización

La transparencia de localización se logra creando un conjunto de seudónimos o alias para cada usuario. Así, el usuario puede referirse a los datos usando nombres sencillos que el sistema traduce a nombres completos.

Con el uso de seudónimos, no será necesario que el usuario conozca la localización física de un dato. Además, el administrador de la base de datos puede cambiar un dato de una localidad a otra sin afectar a los usuarios.

Transparencia y actualizaciones

Es más difícil hacer transparente la base de datos para usuarios que la actualizan que para aquellos que sólo leen datos. El problema principal es asegurarse que se actualizan todas las copias de un dato y también todos los fragmentos afectados.

Procesamiento de consultas

En el caso de sistemas centralizados, el criterio principal para determinar el costo de una estrategia específica es el número de accesos al disco. En un sistema distribuido es preciso tener en cuenta otros factores, como son:

- El costo de transmisión de datos en la red.
- El beneficio potencial que supondría en la ejecución el que varias localidades procesaran en paralelo partes de una consulta.

Recuperación en sistemas distribuidos

Una transacción debe ejecutarse de forma atómica. Además, en el caso de ejecución concurrente, el efecto de ejecutar una transacción debe ser el mismo que si se ejecutara sola en el sistema.

Estructura del sistema

Garantizar la atomicidad es mucho más difícil en un sistema distribuido. Esto se debe a que es posible que participen varias localidades en la ejecución de una transacción. El fallo de una de estas localidades o el fallo de la línea de comunicación entre ellas, puede dar como resultado un cálculo erróneo.

La función del gestor de transacciones de un sistema de base de datos distribuido es asegurar que la ejecución de las distintas transacciones de un sistema distribuido conserve la atomicidad. Cada localidad cuenta con su propio gestor de transacciones. Los distintos gestores de transacciones cooperan para ejecutar las transacciones globales.

Cada localidad del sistema contiene dos subsistemas:

- **Gestor de transacciones**: su función es gestionar la ejecución de aquellas transacciones que accedan a datos almacenados en esa localidad.
- **Coordinador de transacciones**: su función es coordinar la ejecución de varias transacciones iniciadas en esa localidad.

Cada gestor de transacciones se encarga de:

- Mantener una bitácora para la recuperación.
- Participar en un esquema de control de concurrencia apropiado para coordinar la ejecución en paralelo de las transacciones que se ejecuten en esa localidad.

Para cada transacción el coordinador debe:

- Iniciar la ejecución de la transacción.
- Dividir la transacción en varias subtransacciones, las cuales debe distribuir en las localidades apropiadas para su ejecución.
- Coordinar la terminación de la transacción, ya que puede ser terminada o abortada en todas las localidades.

Robustez

Un sistema distribuido puede sufrir el mismo tipo de fallos que un sistema centralizado. Sin embargo, en una configuración distribuida es necesario prever otro tipo de fallos como pueden ser:

- El fallo total de una localidad.
- La interrupción de una línea de comunicación.
- Pérdida de mensajes.
- Fragmentación de la red.

Por lo general, es posible detectar que existe un fallo, pero resulta difícil identificar el tipo del que se trata.

Cuando se detecta un fallo, se debe iniciar un procedimiento de reconfiguración del sistema que le permita continuar con sus operaciones normales.

- Si en la localidad que está fuera de servicio se almacena información repetida, debe actualizarse el catálogo de manera que las consultas no hagan referencia a la copia que se encuentra en dicha localidad.
- Si en el momento de presentarse el fallo existían transacciones activas en la localidad que quedó fuera de servicio, deben abortarse.
- Si la localidad que quedó fuera de servicio es el distribuidor central de algún subsistema, es preciso "elegir" un nuevo distribuidor.

Protocolos de compromiso

Para garantizar la atomicidad, es preciso que todas las localidades en las que se haya ejecutado la transacción T coincidan en el resultado final de la ejecución. T debe quedar ejecutada o abortada en todas las localidades. Para garantizar esta propiedad, el coordinador de transacciones encargado de T debe ejecutar un protocolo de compromiso.

Compromiso de dos fases

Sea T una transacción que se inició en la localidad L_i, y sea C_i el coordinador de transacciones de esa localidad. Cuando T termina de ejecutarse, C_i inicia el protocolo de compromiso de dos fases.

Fase 1: C_i añade el registro <**preparar** T> a la bitácora y la graba en memoria estable. Una vez hecho esto envía un mensaje de **preparar** T a todas las localidades en las que se ejecutó T. Al recibir el mensaje, el gestor de transacciones de cada una de esas localidades determina si está dispuesto a ejecutar la parte de T que le correspondió. Si no está dispuesto, éste añade un registro <**no** T> a la bitácora y envía un mensaje **abortar** T a C_i. Si la respuesta es afirmativa agregará un registro <T **lista**> a la bitácora y grabará todos los registros de bitácora que corresponden a T en memoria estable. Luego, responderá con el mensaje T **lista**.

Fase 2: una vez que todas las localidades hayan respondido al mensaje preparar T enviado a C_i, C_i puede determinar si la transacción T puede ejecutarse o abortarse. Si C_i recibió el mensaje T lista de todas las localidades que participan, la transacción T puede ejecutarse. En caso contrario, la transacción T debe abortarse. Según haya sido el veredicto, se agregará un registro <**ejecutar** T> o <**abortar** T> a la bitácora y se grabará en memoria estable. Luego, el coordinador enviará un mensaje <ejecutar T> o <abortar T> a todas las localidades participantes.

Manejo de fallos

Fallo de una localidad participante

En el momento en que una localidad participante L_k se recupera de un fallo, debe examinar su bitácora para determinar el destino de aquellas transacciones que se estaban ejecutando cuando se produjo el fallo.

- La bitácora contiene un registro <**ejecutar** T>, entonces la localidad ejecuta rehacer(T).
- La bitácora contiene un registro <**abortar** T>, entonces la localidad ejecuta deshacer(T).
- La bitácora contiene un registro <T **lista**>, entonces la localidad debe consultar a C_i para determinar el destino de T. Si C_i está activo, notificará a L_k de la ejecución o aborto de T. Si no está activo, L_k debe interrogar a las demás localidades para intentar determinar el destino de T.
- La bitácora **no contiene registros de control** referentes a T. Esto implica que L_k tuvo un fallo antes de responder al mensaje preparar T de C_i. Puesto que el fallo de L_k excluye el envío de esa respuesta, C_i debe abortar T. En consecuencia, L_k debe ejecutar deshacer(T).

Fallo del coordinador

Si el coordinador falla en la mitad de la ejecución del protocolo de compromiso de la transacción T, entonces las localidades participantes deben decidir el destino de T.

- Si una localidad activa contiene un registro <**ejecutar** T> en su bitácora, entonces T debe ser ejecutada.
- Si una localidad activa contiene un registro <**abortar** T> en su bitácora, entonces T debe ser abortada.
- Si alguna localidad activa no contiene un registro <T **lista**> en su bitácora, entonces el coordinador que fallo C_i no puede haber decidido ejecutar T. Sin embargo, el coordinador puede haber decidido abortar T, y no ejecutar T. Es preferible abortar T antes que esperar que se recupere C_i.
- Si no ocurre ninguno de los casos anteriores, entonces todas las localidades activas deben tener en sus bitácoras un registro <T lista>, y ningún registro de control adicional. En este caso se debe esperar la recuperación de C_i para decidir el futuro de la transacción.

Fallo de una línea de comunicación

Cuando una línea de comunicación falla, todos los mensajes que estaban siendo enviados a través de la línea no llegan a sus destinos intactos. Desde el punto de vista de las localidades conectadas a través de esa línea, parece que el fallo corresponde a otras localidades. Por lo tanto, el esquema anterior también se puede aplicar aquí.

Fragmentación de la red

Cuando se fragmenta la red caben dos posibilidades:

- El coordinador y todos sus participantes quedan en un fragmento. En este caso el fallo no tendrá efecto sobre el protocolo de compromiso.
- El coordinador y sus participantes quedan distribuidos en varios fragmentos. En este caso se perderán los mensajes entre la participante y el coordinador.

Compromiso de tres fases

El protocolo de compromiso de tres fases está diseñado para impedir la posibilidad de bloqueo en un caso de los fallos posibles. El protocolo requiere que:

- No pueda ocurrir una fragmentación de la red.
- Debe haber al menos una localidad funcionando en cualquier punto.
- En cualquier punto, como máximo un número K de participantes pueden caer simultáneamente (siendo K un parámetro que indica la resistencia del protocolo a fallos en localidades).

El protocolo alcanza esta propiedad de no bloqueo añadiendo una fase extra, en la cual se toma una decisión preliminar sobre el destino de T. Como resultado de esta decisión, se pone en conocimiento de las localidades participantes cierta información, que permite tomar una decisión a pesar del fallo del coordinador.

Sea T una transacción iniciada en la localidad L_i y C_i el coordinador de transacciones en L_i.

Fase 1: es igual a la fase 1 del protocolo de compromiso de dos fases.

Fase 2: si C_i recibe un mensaje abortar T de una localidad participante, o si C_i no recibe respuesta dentro de un intervalo previamente especificado de una localidad participante, entonces C_i decide abortar. La decisión de abortar está implementada de la misma forma que en el protocolo de dos fases. Si C_i recibe un mensaje T lista de cada localidad participante, tomará la decisión preliminar de "**preejecutar**" T. La diferencia entre preejecutar y ejecutar radica en que T puede ser todavía abortado eventualmente. La decisión de preejecutar permite al coordinador informar a cada localidad participante que todas las localidades participantes están listas. C_i añade un registro <preejecutar T> a la bitácora y lo graba en un almacenamiento estable. C_i envía un mensaje preejecutar T a todas las localidades participantes. Cuando una localidad recibe un mensaje del coordinador, lo registra en su bitácora, grabando esta información en almacenamiento estable, y envía un mensaje de reconocimiento a T al coordinador.

Fase 3: esta fase se ejecuta sólo si la decisión tomada en la fase 2 fue de preejecutar. Después de que los mensajes preejecutar T se han enviado a todas las localidades participantes, el coordinador debe esperar hasta que reciba al menos un número K de mensajes de reconocimiento a T. Siguiendo este proceso, el coordinador toma una decisión de compromiso. Añade un registro <ejecutar T> en su bitácora y la graba en memoria estable. C_i envía un mensaje ejecutar T a todas las localidades participantes. Cuando una localidad recibe el mensaje, lo registra en su bitácora.

Manejo de fallos

Fallo de una localidad participante

En el momento en que una localidad participante L_k se recupera de un fallo, debe examinar su bitácora para determinar el destino de aquellas transacciones que se estaban ejecutando cuando se produjo el fallo.

- La bitácora contiene un registro <**ejecutar** T>, entonces la localidad ejecuta rehacer(T).
- La bitácora contiene un registro <**abortar** T>, entonces la localidad ejecuta deshacer(T).
- La bitácora contiene un registro <T **lista**> y ningún <**abortar** T> o <**ejecutar** T>, entonces la localidad debe consultar a C_i para determinar el destino de T. Si C_i responde con un mensaje preejecutar T, la localidad lo registra en su bitácora y continúa el protocolo enviando un mensaje de reconocimiento a T al coordinador. Si C_i responde con un mensaje que se ha ejecutado T, la localidad ejecutará rehacer(T). En el caso de que C_i falle al responder dentro de un intervalo previamente especificado, la localidad ejecutará un protocolo de fallo del coordinador.
- La bitácora contiene un registro <preejecutar T>, y ningún registro <abortar T> o <ejecutar T>. Como en el caso anterior, la localidad debe consultar con C_i.

Fallo del coordinador

Si una localidad participante falla al recibir una respuesta del coordinador, ésta ejecuta el protocolo coordinador de fallos. Este protocolo resulta en la selección de un nuevo coordinador. Cuando el coordinador que falló se recupera, toma el papel de una localidad participante.

Protocolo de fallo del coordinador

1) Las localidades participantes activas eligen un nuevo coordinador utilizando un protocolo de selección.
2) El nuevo coordinador, C_{nuevo}, envía un mensaje a cada localidad participante pidiendo el estado local de T.
3) Cada localidad participante, incluyendo C_{nuevo}, determina el estado local de T y envía su estado local a C_{nuevo}.
4) Dependiendo de la respuesta recibida, C_{nuevo} decide ejecutar o abortar T o reiniciar el protocolo de compromiso de tres fases.

La consideración de no aceptar una fragmentación de la red es crucial. La fragmentación de la red podría conducirnos a la elección de dos nuevos coordinadores, cuyas decisiones no podrían coincidir.

La consideración de que no todas las localidades fallen a la vez, es también crucial. Si todas las localidades fallan, sería solo la última en caer la que puede tomar una decisión. Esto nos llevaría a un problema de bloqueo, puesto que las demás localidades tienen que esperar a que esta última se recupere. Además, es muy difícil determinar cual fue la última en caer después del fallo de todas las localidades.

Finalmente es, muy importante la elección del parámetro K, dado que si estuvieran activos menos de un número K de participantes, se produciría un bloqueo.

Comparación de protocolos

A pesar de la posibilidad de bloqueo, el protocolo de compromiso de dos fases es utilizado más comúnmente. La probabilidad de que en la práctica ocurra un bloqueo es normalmente lo suficientemente baja para que no este justificado el coste extra que supone el compromiso de tres fases. Además, el compromiso de tres fases es muy vulnerable a la hora de enlazar los fallos. Esta desventaja puede ser salvada con protocolos de nivel de red, pero de esta forma se aumenta el tiempo extra.

Control de concurrencia - Protocolos de bloqueo

Prácticamente todos los protocolos de bloqueo que se describieron pueden aplicarse en una configuración distribuida. Lo único que es necesario cambiar es la forma en que se estructura el gestor de bloqueos.

Se partirá del supuesto de que existen los modos de bloqueo compartido y exclusivo.

Esquema sin repetición

Cada localidad mantiene un gestor de bloqueo local cuya función es la de administrar las solicitudes de bloqueo y desbloqueo para aquellos datos que estén almacenados en esa localidad. Cuando una transacción desea bloquear el dato Q en la localidad L_i, enviará un mensaje al gestor de bloqueos de dicha localidad solicitando un bloqueo.

Este esquema es fácil de implementar. Requiere el envío de dos mensajes para el manejo de la petición de bloqueo y de uno para la petición de desbloqueo. Sin embargo, el manejo de bloqueos es más complejo.

Enfoque de coordinador único

El sistema mantiene un único gestor de bloqueos que reside en una única localidad L_i, elegida para tal propósito. Todas las solicitudes de bloqueo y desbloqueo se hacen en la localidad L_i.

Ventajas:

- Sencillez de implementación: requiere dos mensajes para manejar solicitudes de bloqueo y de uno para atender solicitudes de desbloqueo.
- Sencillez del manejo de bloqueos.

Desventajas:

- Cuello de botella: dado que todas las solicitudes deben procesarse en la localidad L_i, está se convertirá en un cuello de botella.
- Vulnerabilidad: si la localidad L_i queda fuera de servicio, se perderá el controlador de concurrencia. Será necesario detener el procesamiento o utilizar un esquema de recuperación.

Puede llegarse a un equilibrio entre las ventajas y las desventajas por medio de un enfoque de **coordinadores múltiples**, en el cual la función de manejo de bloqueos se distribuye entre varias localidades.

Cada uno de los gestores administra las solicitudes de bloqueo y desbloqueo para un subconjunto de datos. Cada gestor de bloqueos reside en una localidad diferente. Esto reduce la posibilidad de que se produzca un cuello de botella en el coordinador, pero complicaría el manejo de bloqueos, dado que las solicitudes de bloqueo y desbloqueo no se atienden en la misma localidad.

Protocolo de mayoría

El sistema mantiene un gestor de bloqueos en cada una de las localidades. Cada gestor administra los bloqueos de todos los datos o copias de datos almacenados en esa localidad. Cuando una transacción desea poner un bloqueo al dato Q que se repite en n localidades diferentes, debe enviar una solicitud de bloqueo a más de la mitad de las n localidades en las que existe una copia de Q. La transacción no operará sobre Q hasta que haya logrado que se coloquen bloqueos en más de la mitad de las copias de Q.

Este esquema tiene la ventaja de que los datos repetidos se manejan de forma descentralizada, lo cual evita los problemas de control central. Sin embargo, tiene las siguientes desventajas:

- Realización: son necesarios 2(n/2 + 1) mensajes para manejar solicitudes de bloqueos y 2(n/2 + 1) mensajes para atender solicitudes de desbloqueos.
- Manejo de bloqueos: es posible que se presente un bloqueo aun en el caso de que se coloque un bloqueo a un solo dato.

Protocolo preferencial o sesgado

El sistema mantiene un gestor de bloqueos en cada una de las localidades. Cada gestor administra los bloqueos de todos los datos almacenados en esa localidad. Los bloqueos compartidos y los exclusivos son manejados de forma diferente:

- Bloqueos compartidos: cuando una transacción necesita bloquear el dato Q, lo único que tiene que hacer es solicitarlo al gestor de bloqueos de cualquier localidad que contenga una copia de Q.
- Bloqueos exclusivos: cuando una transacción necesita bloquear el dato Q, debe solicitarlo al gestor de bloqueos de todas las localidades que contengan una copia de Q.

Este esquema tiene la ventaja de que las operaciones de lectura consumen menos tiempo extra que con el protocolo de mayoría. Sin embargo, tiene la desventaja del tiempo extra de las operaciones de escritura. Además, comparte con el protocolo de mayoría la desventaja de hacer más complejo el manejo de bloqueos.

Copia primaria

En el caso de que exista repetición de datos, es posible elegir una de las copias como copia primaria. Así, para cada dato Q, la copia primaria de Q debe residir en una localidad determinada, llamada **localidad primaria** de Q.

Cuando una transacción necesita bloquear el dato Q, lo solicita en la localidad primaria de Q.

Si la localidad primaria de Q queda fuera de servicio, no se podrá tener acceso a Q aunque existan otras localidades accesibles que contengan una copia.

Asignación de horas de entrada

Existen dos métodos principales para generar horas de entrada únicas, uno centralizado y otro distribuido. En el esquema centralizado se elige una sola localidad para que se encargue de distribuir las horas de entrada.

En el esquema distribuido, cada una de las localidades genera una hora de entrada única, ya sea al utilizar un contador lógico o el reloj local. Las horas de entrada global única se obtienen concatenando las horas de entrada local con el identificador de la localidad, que debe ser único. El identificador de localidad

se coloca en la posición menos significativa para garantizar que las horas de entrada globales que se generen en una localidad no van a ser siempre mayores que las que se generen en otra localidad.

Se necesita un mecanismo para garantizar que las horas de entrada locales se generen en forma pareja en todo el sistema. Para esto, se define un reloj lógico dentro de cada localidad que se encargará de generar la hora de entrada local única.

Si se utiliza el reloj del sistema para generar horas de entrada, la asignación de horas de entrada será pareja siempre que ninguno de los relojes se adelante o se atrase.

Manejo de bloqueos

Los algoritmos de prevención y detección de bloqueos pueden aplicarse en un sistema distribuido.

La prevención de bloqueos puede hacer que algunas transacciones esperen y otras retrocedan sin que sea realmente necesario.

En todos los esquemas se requiere que cada localidad mantenga un grafo de espera local. Los nodos del grafo corresponden a todas las transacciones (locales o no) que en un momento dado utilizan o solicitan algunos de los datos que pertenecen a esa localidad.

Si cualquier grafo de espera local contiene un ciclo, se habrá presentado un bloqueo. Sin embargo, el hecho de que ninguno de los grafos locales de espera contenga un ciclo no significa que no existan bloqueos. El bloqueo existe cuando al realizar la unión de los grafos locales de espera contiene un ciclo.

Selección del coordinador

Si el coordinador falla al quedar fuera de servicio, la localidad donde reside, el sistema podrá continuar la ejecución sólo si activa un coordinador en alguna otra localidad. Esto puede hacerse manteniendo una copia de seguridad del coordinador o eligiendo un nuevo coordinador después de que haya fallado el coordinador original.

Coordinadores de copia de seguridad (back-up)

Un coordinador de copia de seguridad es una localidad que, además de realizar otras tareas, mantiene de manera local la información que le permite asumir el papel de coordinador con una dislocación mínima del sistema distribuido. Tanto el coordinador como su copia de seguridad reciben todos los mensajes dirigidos al coordinador. La única diferencia entre el coordinador y su copia de seguridad es que este último no emprende acciones que afecten a otras localidades. Tales acciones se dejan al coordinador real.

Cuando el coordinador de copia de seguridad se da cuenta que el coordinador está fuera de servicio asume el papel de coordinador. Dado que la copia de seguridad dispone de toda la información que disponía el coordinador inactivo, el procesamiento puede continuar sin interrupciones.

La ventaja principal de este enfoque es que el procesamiento puede reanudarse inmediatamente. Si una copia de seguridad no estaba lista para asumir la responsabilidad del coordinador, se elegirá uno nuevo, el cual tendrá que pedir información a todas las localidades del sistema para poder ejecutar las tareas de coordinación.

La desventaja de este enfoque es el tiempo extra que requiere la doble ejecución de las tareas del coordinador. Además, es necesario que el coordinador y su copia de seguridad se comuniquen de forma regular para garantizar la sincronización de sus actividades.

Algoritmos de elección

Los algoritmos de elección requieren que se asigne un número de identificación único a cada localidad activa del sistema. Para hacer más sencillo el análisis, se supondrá que el coordinador reside siempre en la localidad que cuenta con el número de identificación más alto. El objetivo de un algoritmo de elección es escoger la localidad del nuevo coordinador. Por lo tanto, cuando un coordinador queda fuera de servicio, el algoritmo deberá escoger la localidad activa que tenga el número de identificación más alto. Este número debe comunicarse a todas las localidades activas del sistema. Además, el algoritmo debe incluir un mecanismo para que una localidad que se esté recuperando de un fallo pueda identificar al coordinador actual.

Existen varios algoritmos de elección diferentes. La diferencia radica normalmente en la configuración de la red.

Unidad 9

Nuevas aplicaciones de las bases de datos

Hasta ahora los modelos de datos tienen las siguientes características:

- Uniformidad: existe un gran número de datos estructurados de manera similar, todos del mismo tamaño.
- Orientación en registros: los datos básicos constan de registros de longitud fija.
- Datos pequeños: todos los registros son cortos.
- Campos atómicos: los campos de un registro son cortos y de longitud fija. No hay una estructura en los campos. En otras palabras, se cumple la primera forma normal.
- Transacciones cortas: no hay interacción humana con la transacción durante su ejecución. En vez de ello, el usuario prepara la transacción, la somete a ejecución y espera la respuesta.
- Esquema de concepto estático: el esquema de la base de datos se cambia con muy poca frecuencia.

Sin embargo, la tecnología de bases de datos se ha adaptado a aplicaciones fuera del ámbito del procesamiento de datos. Estas nuevas aplicaciones incluyen:

- Diseño asistido por computador (CAD).
- Ingeniería de software asistida por computador (CASE).
- Bases de datos de multimedios.
- Sistemas de información de oficina.
- Sistemas expertos de bases de datos.

Estas nuevas aplicaciones requieren nuevos modelos de datos, nuevos lenguajes de consulta y nuevos modelos de transacciones. Entre los requisitos de estas nuevas aplicaciones están:

- Objetos complejos: un objeto complejo es un dato que es visto como un simple objeto en el mundo real, pero que contiene otros objetos. Estos objetos pueden tener una estructura interna compleja arbitraria.
- Datos de comportamiento: puede que los objetos necesiten responder de diferentes formas a la misma orden. Esta información sobre el comportamiento puede capturarse almacenando código ejecutable con objetos en la base de datos.
- Meta conocimiento: a veces los datos más importantes sobre aplicaciones son reglas generales acerca de la aplicación más que de las tuplas específicas.
- Transacciones de larga duración.

Estructura de objetos

El modelo orientado a objetos se basa en encapsular código y datos en una única unidad, llamada **objeto**.

En general, un objeto tiene asociado:

- Un conjunto de **variables** que contienen los datos del objeto. El valor de cada variable es un objeto.
- Un conjunto de **mensajes** a los que el objeto responde.
- Un **método**, que es un trozo de código para implementar cada mensaje. Un método devuelve un valor como respuesta al mensaje.

El termino mensaje en un contexto orientado a objetos se refiere al paso de solicitudes entre objetos sin tener en cuenta detalles específicos de implementación.

Dado que la única interfaz externa que representa un objeto es el conjunto de mensajes al que responde, es posible modificar la definición de métodos y variables sin afectar a otros objetos. También es posible sustituir una variable por un método que calcule un valor.

Jerarquía de clases

Normalmente en una base de datos existen muchos objetos que responden a los mismos mensajes, utilizan los mismos métodos y tienen variables del mismo nombre y tipo. Sería un trabajo inútil definir cada uno de estos objetos por separado. Por tanto, agrupamos los objetos similares para que formen una **clase**.

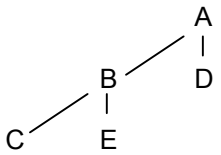
A cada uno de estos objetos se le llama instancia de su clase. Todos los objetos de una clase comparten una definición común, aunque difieran en los valores asignados a las variables.

Un objeto clase incluye:

- Una variable con valores en un conjunto cuyo valor es el conjunto de todos los objetos que son instancias de la clase.
- Implementación de un método para el nuevo mensaje, el cual crea una nueva instancia de la clase.

Un esquema de base de datos orientada a objetos normalmente requiere un gran número de clases. Sin embargo, a menudo se da el caso de que varias clases son similares.

Para permitir la representación directa de similaridades entre clases necesitamos colocar clases en una jerarquía de especialización como la que se definió para el modelo entidad-relación.



Un objeto que representa a A contiene todas las variables de las clases A, B y C. Esto se refiere a la **herencia** de las propiedades de una clase más general. Los métodos se heredan de la misma manera que las variables.

Herencia múltiple

El concepto de herencia múltiple se refiere a la capacidad de las clases para heredar variables y métodos de múltiples superclases. La relación clase/subclase se representa por un grafo con raíz acíclica dirigido en el que una clase puede tener más de una superclase.

Cuando se emplea herencia múltiple es posible que se de ambigüedad en el caso en que pueda heredarse la misma variable o método de más de una superclase. En este caso se debe optar por:

- Incluir las dos variables y renombrarlas como A.variable y B.variable.
- Elegir una u otra basándose en el orden en que se crearon las clases.
- Obligar al usuario a que haga la elección en el momento en que se define la clase.

Identidad de objetos

Un objeto conserva su identidad aun cuando algunos o todos los valores de las variables o las definiciones de los métodos cambien con el tiempo.

La identidad de objeto es una noción más fuerte que la que se encuentra normalmente en los lenguajes de programación o en los modelos de datos que no están basados en la orientación a objetos.

Hay varias formas de identidad:

- Valor: se utiliza un valor de dato por identidad.
- Nombre: se utiliza un nombre facilitado por el usuario por identidad. Esta es la forma de identidad que normalmente se usa para las variables en los procedimientos. A cada variable se le da un nombre que identifica de manera única a la variable sin importar el valor que contenga.
- Incorporación: una noción de identidad es incorporar en el modelo de datos el lenguaje de programación, y no se requiere que el usuario proporcione ningún identificador. Esta es la forma de identidad que se usa en los sistemas orientados a objetos.

Grados de permanencia de la identidad:

- Intraprograma: la identidad permanece solamente durante la ejecución de un único programa o consulta.
- Interprograma: la identidad permanece de una ejecución del programa a otra.
- Persistente: la identidad permanece no solo entre las ejecuciones del programa sino también entre las reorganizaciones estructurales de los datos. Esta forma es la que se requiere en los sistemas orientados a objetos.

Sistema de ayuda a las decisiones

Data warehouse: (almacén, repositorio de datos)

- Un lugar donde las personas pueden acceder a sus datos.
- Una base de datos que contiene los datos históricos producidos por los sistemas informáticos de las empresas.
- Son datos administrados que están situados fuera y después de los sistemas operacionales.
- Se remite al adecuado manejo de la información histórica generada por una organización, a fin que esta pueda otorgarle a sus ejecutivos las herramientas adecuadas para el proceso de toma de decisiones.

DW un tema de negocios:

- *Cuando las necesidades del mercado y los progresos tecnológicos convergen, se dan los cambios más importantes en las prácticas de negocios.*
- Antecedentes:
    - Evolución de las clases de aplicaciones en las organizaciones.
    - Evolución de la tecnología de software y hardware.
    - Cambios en la naturaleza de los negocios (globalización).
- Casos que se presentan:
    - Tenemos montañas de datos pero no tenemos acceso a ellos.
    - Todos saben que algunos datos no son confiables.
    - Solo mostrame lo que es importante.

Nada enloquece más a los gerentes que dos personas presentando el mismo resultado de negocios pero con números distintos.

- Objetivos:
    - Brindar acceso a los datos (eficiente, fácil, consistente, confiable).
    - Brindar soporte a la toma de decisiones, por ejemplo permitiendo realizar predicciones de operaciones futuras de un modo racional.
    - Encontrar relaciones entre los datos producidos por diferentes áreas, determinando patrones que permitan predecir futuros sucesos.
    - Identificar nuevas oportunidades de negocio.
    - Monitorear las operaciones actuales de negocio y compararlas con las operaciones efectuadas previamente.
    - Reducción de los costos de gestión de la información.
    - Aumento de la productividad.
    - Posibilidad de separar y combinar los datos por medio de toda posible medida en el negocio. Por ejemplo, para análisis del problema en términos de dimensiones.
    - Reducción de los costos de gestión de información.

Si el DW no logra transformar los datos en información para hacerla accesible a los usuarios en un tiempo lo suficientemente rápido como para marcar una diferencia, no sirve.

En cuanto a…	Mundo operacional. Transacción	Mundo analítico. DW
Objetivos	Mantienen la empresa en operación. ABM y C. Diseño de bases de datos y procesos.	Brindan lo necesario para el planeamiento y toma de decisiones. Solo modelo de datos.
Requisitos	Necesidad de performance.	Necesidad de flexibilidad y alcance amplio.
Datos almacenados	Son para propósito operacional. Datos volátiles, se modifican. Contienen datos de períodos acotados de tiempo por cuestiones de performance. Los datos están encriptados por restricciones del lenguaje o de la base de datos (normalizaciones).	Son para propósito analítico. Datos no volátiles. Pueden mantenerse por tiempo indefinido. NO implica más costo. Los datos se hallan expresados en términos simples de negocio.

En cuanto a…	Mundo operacional. Transacción	Mundo analítico. DW
Datos que almacenan	Los datos giran alrededor de aplicaciones y procesos. No siempre están integrados (aunque deberían). Los datos deberían estar normalizados. Generalmente no hay necesidad de hacer referencia al tiempo en las estructuras claves.	Los datos giran alrededor de los sujetos de la empresa. Están fuertemente integrados para que el análisis tenga sentido. Los datos pueden no estar normalizados, es conveniente. Toda estructura clave contiene un elemento de tiempo. Puede tener datos externos (mercado, tendencia).
Operaciones/ transacciones	Las transacciones están predefinidas. Tienen uso habitual. Operaciones: ABM Queries tradicionales SQL.	Transacciones difíciles de predecir, nunca se sabe que se pedirá. Operaciones: carga inicial y acceso a datos (no modificaciones). Análisis multimedial para consultas.

Definiciones

- Es un ambiente estructurado y extensible diseñado para el análisis de datos no volátiles, transformados lógica y físicamente desde varias aplicaciones fuentes para alinearse con la estructura de negocio, actualizado y mantenido por un período de tiempo largo, expresado en término de negocio simples y resumido para un análisis rápido.
- Es una colección de datos orientada a sujetos, integrada, variante en el tiempo, no volátil para soportar el proceso de toma de decisiones.
- Es un proceso y no un producto. Es una técnica de ensamblar y gestionar adecuadamente los datos procedentes de distintas fuentes con el objeto de tener una visión única de los mismos para toda la empresa.

Componentes

- Bases de datos fuentes (producción).
- Bases de datos con datos resumidos (DW).
- Interfaces de acceso (consultas, reportes, análisis multidimensional, Data Mining).

Elementos que participan en la construcción de un DW

- Recursos humanos.
- Metodología de trabajo.
- Herramientas de hardware y software.
- Infraestructura de la organización en general.

Etapas de diseño de un DW

- Análisis:
    - Necesidad de la organización.
    - Áreas involucradas.
    - Evaluación de herramientas y arquitectura tecnológica.
- Diseño del modelo de datos:
    - Forma de manipular la información por cada área de la organización.
    - Compatibilización de modelos.
    - Generación de la estructura que sirva como modelo de datos.
- Selección y extracción de datos:
    - Información transaccional (operatividad).
    - Actualización de datos vs. Extracción.
    - Performance (transacciones en línea).
    - Creación de vistas (espacio adicional vs. Mejor tiempo de acceso).
- Limpieza y transformación:
    - Calidad de datos de fuentes diferentes.

    - Técnicas de limpieza (errores inherentes en los sistemas, errores de validación de dominios, tipeos incorrectos).
    - Data scrubbing: proceso de filtrado, decodificación y traducción de datos fuentes para crear datos validos.
    - Homogeneizar tipos de datos.
    - Valores por defecto inteligentes.
    - Períodos de tiempo completos.
- Creación de los metadatos:
    - Proveen información sobre los datos almacenados en el DW
- Carga:
    - Preprocesamiento adicional.
    - Checkpoint.
    - Secuencias o paralelo.
- Actualización:
    - Periodicidad.
    - Cumplimiento de etapas con los nuevos datos.

Data Marts

Satisfacen necesidades específicas de un área. Pueden ser un subconjunto de un DW.
Pueden coexistir varios DM dentro de una empresa, cada área de la misma puede tener uno.

Data Marts	Data Warehouse
Enfocado a una sola área.	Contiene información de toda la organización.
Menor costo.	Mayor costo.
Puede existir más de un DM.	Existe solo uno en toda la organización.
Fácil de entender y acceder.	Más complejo.
Mejor tiempo de acceso y respuesta.	Menor performance.
A mayor número de DM, administración más costosa.	Administración más eficiente.
Peligro de crear ”islas” difíciles de integrar.	Integrado por naturaleza.
Peligro de duplicación de datos en varios DM.	No existe duplicación de datos.

Data Mining

Extracción no trivial de información implícita previamente desconocida y potencialmente útil de la base de datos.

Los datos ocultan desviaciones, tendencias, anomalías, que se descubren con reglas de inducción, redes neuronales, etc.

Ventajas

- Ayuda a soportar el proceso de toma de decisiones.
- Identificación de nuevas oportunidades en el mercado.
- Estructuras corporativas flexibles: distintos departamentos en una organización comparten la misma información.
- Generación de informes críticos sin costo de tiempo.
- Formato de datos consistente.
- Servir de plataforma efectiva de mezclado de datos desde varias aplicaciones corrientes.
- Posibilidad de encontrar respuestas más rápidas y más eficientes a las necesidades del momento.
- Reducción de costos globales para la organización.

Conclusiones

- La implantación de un DW surge como la solución a la necesidad de sistemas de información que permitan generar consultas, reportes y análisis para la toma de decisiones en cada día más apremiante para las empresas que quieren competir exitosamente en el mercado.
- Hoy en día desarrollar un DW es raramente rápido y fácil. Pocas instalaciones toman menos de seis meses (la mayoría toma dos años o más).
- No siempre en las organizaciones existe una conciencia real de la potencialidad que tienen encubierta en los datos transaccionales generados cotidianamente.
- Las organizaciones de tecnología informática pueden o no tener todas las cualidades técnicas necesarias, pero no implementarán un proyecto de DW exitoso sin que logren que la unidad de negocios se involucre.
- Se debe tener en cuenta que un DW no es un producto que se compra, debe ser planificado cuidadosamente de acuerdo a cada organización y construirlo a tal efecto.
- Cuando se diseña un DW hay que tener en claro que se está convirtiendo la información generada en forma cotidiana sin conocimiento para la organización.
- Pensar en un DW como una simple liberación de los datos corporativos sería un error. El valor real de un DW recién se descubre cuando lo utiliza alguien que puede encontrar detalles importantes en los datos y marcar las diferencias.

Unidad 10

Almacenamiento primario y secundario

El almacenamiento primario (RAM) tiene las siguientes desventajas:

- Existe un límite en la cantidad de memoria RAM accesible en un computador determinado.
- Es un dispositivo relativamente caro.
- Cuando se apaga la computadora desaparece todo lo que estaba almacenado en la memoria RAM.

El almacenamiento secundario se refiere a los medios de almacenamiento que están fuera del almacenamiento primario en memoria RAM dentro de la computadora. Con el almacenamiento secundario se tienen las siguientes ventajas:

- Almacenar más información que lo que se puede almacenar en la memoria RAM.
- Son más económicos.
- No requieren el suministro continuo de energía para conservar la información almacenada.

El almacenamiento secundario tiene como desventaja el hecho de que es más lento el acceso a la información.

Archivos

Un **archivo** es una colección de información relacionada, de acuerdo a las siguientes definiciones:

- Colección de registros que abarca un conjunto de entidades con aspectos en común organizados para un propósito en particular.
- Colección de registros semejantes que se guardan en almacenamiento secundario del computador.
- Colección de datos almacenados en dispositivos secundarios de memoria.

Estructura de archivos

La **estructura de un archivo** es la organización impuesta a un archivo para facilitar su procesamiento. Las estructuras de archivos incluyen campos, registros, bloques, árboles, índices, secuencias y otras construcciones conceptuales.

Archivos físicos y lógicos

Un **archivo físico** es un archivo que en realidad existe en el almacenamiento secundario. Es el archivo tal como lo conoce el sistema operativo y que aparece en el directorio de archivos.

Un **archivo lógico** es el archivo visto por el programa. El uso de archivos lógicos permite a un programa describir las operaciones que van a efectuarse en un archivo sin saber cual es el archivo físico real que se usará. El programa puede entonces usarse para procesar cualquiera de diversos archivos que comparten la misma estructura.

Operaciones básicas

La **apertura** de un archivo implica que está listo para que el programa lo use. El usuario estará colocado al principio del archivo y listo para leer o escribir. El contenido del archivo no se altera.

Crear un archivo es también abrirlo, en el sentido de que estará listo para usarse después de creado. Dado que un archivo recién creado no tiene contenido, el único uso que tiene sentido al inicio es la escritura.

Cuando se **cierra** un archivo, el nombre lógico del archivo queda disponible para usarse con otro archivo. El cierre de un archivo que se ha usado para salida también asegura que todo se ha escrito en el archivo.

Por lo general, el sistema operativo cierra los archivos automáticamente cuando el programa termina.

La **lectura** y **escritura** son fundamentales para el procesamiento de archivos, ya que son acciones que efectúan las operaciones de E/S. Una llamada de lectura de bajo nivel requiere tres componentes de información: un nombre lógico de archivo fuente que corresponda a un archivo abierto, la dirección destino de los bytes que se leerán y el tamaño o cantidad de datos que se van a leer.

Una llamada de escritura de bajo nivel requiere tres componentes de información: un nombre de archivo destino que corresponda a un archivo abierto, la dirección fuente de los bytes que serán escritos y el tamaño o cantidad de datos que se van a escribir.

La función **eof**() pregunta al sistema si el apuntador de lectura/escritura ya pasó del último elemento del archivo. Si esto sucede, eof() devuelve true, sino devuelve false. En el caso de un archivo vacío, eof() devuelve inmediatamente true y no lee ningún byte.

La acción de moverse directamente hasta cierta posición en un archivo suele llamarse **localización** (seek). Esto requiere al menos dos datos: el nombre lógico del archivo donde ocurre la localización y el número de posiciones que el apuntador se mueve desde el inicio del archivo.

El viaje de un byte

El viaje que realiza un byte cuando se envía desde la memoria RAM hacia el disco implica la participación de diversos programas y dispositivos:

- El programa de usuario, que hace la llamada inicial al sistema operativo.
- El administrador de archivos del sistema operativo.
- Un procesador de E/S y su software, que sincronizan la transmisión de un byte entre un buffer de E/S en memoria RAM y el disco.
- El controlador del disco y su software, que dan instrucciones a la unidad acerca de cómo encontrar la pista y el sector apropiados, para después enviar el byte.
- La unidad de disco, que recibe el byte y lo deposita en la superficie del disco.

Administrador de archivos

El **administrador de archivos** puede verse como un conjunto de capas de procedimientos, donde las capas superiores tratan principalmente con los aspectos lógicos de la administración de archivos, y las capas inferiores con los aspectos físicos.

El administrador de archivos comienza por cerciorarse de que las características lógicas del archivo sean compatibles con lo que se le solicita.

Una vez que se identifico el archivo deseado y se verificó la legalidad del acceso solicitado, el administrador debe determinar donde se guardará el dato. Esta información se obtiene de la tabla de asignación de archivos (FAT).

Capas del protocolo de transmisión

- El programa pide al sistema operativo la escritura.
- El sistema operativo transfiere el trabajo al administrador de archivos.
- El administrador busca el archivo en su tabla de archivos y verifica las características.
- Obtiene de la FAT el último sector del archivo.
- El administrador se asegura que es el último sector y graba el dato donde vaya.
- El administrador de archivos da instrucciones al procesador de E/S (donde está el buffer (RAM) y donde queda (Disco))
- El procesador de E/S encuentra el momento para transmitir el dato, la CPU se libera.
- El procesador de E/S envía el dato al controlador de disco (con la dirección de escritura).
- El controlador prepara la escritura y transfiere el dato.

Organización interna de los archivos

Cuando el archivo se almacena como una **secuencia de bytes**, una vez que la información se junta como secuencia de bytes, no se puede volver a separar. No se puede saber cuál es el principio y el final de cada dato.

Un **campo** es la unidad de información lógicamente significativa más pequeña en un archivo. La identidad de los campos se mantiene:

- Forzar a que los campos tengan una longitud predecible: se puede recuperar el archivo con sólo contar hasta el final del campo. Una desventaja es que el archivo crece mucho, además se pueden presentar problemas con datos que sean tan grandes que no quepan en el espacio asignado.
- Comenzar cada campo con un identificador de longitud: si los campos no son muy largos, es posible almacenar la longitud en un solo byte al inicio de cada campo.
- Colocar un delimitador al final de cada campo para separarlo del siguiente: la elección del carácter delimitador es muy importante, ya que debe ser un carácter que no se confunda con lo que se está procesando.

Un **registro** es un conjunto de campos agrupados bajo la perspectiva de un archivo de un nivel más alto de organización en un archivo. Algunos de los métodos que se usan para organizar un archivo en registros son:

- Exigir que los registros sean de longitud predecible. Esta longitud puede medirse en términos de bytes o en términos del número de campos.
- Comenzar cada registro con un indicador de longitud que señale el número de bytes que contiene.
- Usar un segundo archivo para mantener información de la dirección del byte de inicio de cada registro.
- Colocar un delimitador al final de cada registro, para separarlo del siguiente.

Registros de longitud predecible

Hacer un registro de longitud predecible facilita el conteo dentro del registro. El conteo se utiliza para saber si se ha leído todo el registro. El conteo puede hacerse por byte o por campos.

Conteo de bytes: Un archivo con **registros de longitud fija** es aquel cuyos registros contienen todos el mismo número de bytes. Fijar el número de bytes en un registro no implica que los tamaños o el número de campos deban ser fijos. Los registros de longitud fija se usan con mucha frecuencia como recipientes para guardar un número variable de campos de longitud variable.

Campos de conteo: En lugar de especificar que cada registro en un archivo contiene un número fijo de bytes, se puede especificar que contendrá un número fijo de campos.

Comenzar cada registro con un indicador de longitud

Se puede transmitir la longitud de los registros, y hacerlos reconocibles, comenzando cada registro con un campo que contenga un número entero que indique cuantos bytes hay en el resto del registro. Este es un método muy común en el manejo de registros de longitud variable.

Usar un segundo archivo para mantener información sobre las direcciones

Se puede emplear un segundo archivo de índice para mantener información sobre la distancia en bytes de cada registro en el archivo original. La distancia en bytes permite encontrar el comienzo de cada registro sucesivo y calcular su longitud. Se busca la posición de un registro en el índice y después se alcanza el registro dentro del archivo de datos.

Colocar un delimitador al final de cada registro

Esta opción, en el nivel de registro, es similar a la solución que se dio para distinguir los campos.

Extracción de registros por llave: formas canónicas para llaves

Una **forma canónica** para una llave de búsqueda es la representación única para esa llave que se ajusta a la regla.

Una llave no tiene que corresponder a un solo campo en un registro, es posible construir llaves que combinen información de más de un campo del archivo.

Si no hay una relación uno a uno entre la llave y un registro, el programa tendría que proporcionar mecanismos adicionales que permitan al usuario resolver la confusión que puede haber cuando existan más registros que concuerden con una llave en particular.

La solución más sencilla es prevenir tal confusión. La prevención se lleva a cabo cuando se agregan nuevos registros al archivo.

Este requisito de unicidad se aplica solo a las llaves primarias. Una **llave primaria** es la llave que se usa para identificar unívocamente un registro. También es posible, buscar con **llaves secundarias**.

Búsqueda secuencial

La cantidad de trabajo requerido por una búsqueda secuencial es directamente proporcional al número de registros del archivo.

En general, el trabajo requerido para buscar en forma secuencial un registro en un archivo con n registros es proporcional a n. Se dice que una búsqueda secuencial es de orden O(n) porque el tiempo que tarda es proporcional a n.

Manejo de registros en bloques

Existe la posibilidad de mejorar la eficiencia de la búsqueda secuencial leyendo un bloque de varios registros a la vez y después procesar ese bloque de registros en memoria RAM.

Mientras que los campos y registros son formas de mantener la organización lógica dentro del archivo, el manejo de bloques es únicamente una medida para mejorar el desempeño. Como tal, el tamaño del bloque tiene más relación con las propiedades físicas de la unidad de disco que con el contenido de los datos.

El manejo de registros en bloques no cambia el número de comparaciones que deben hacerse en memoria RAM, y es muy probable que incremente la cantidad de datos transferidos entre el disco y la memoria RAM.

El manejo de bloques ahorra tiempo, ya que disminuye el número de desplazamientos del brazo del disco.

Acceso directo

Se tiene **acceso directo** a un registro cuando es posible colocarse directamente en el inicio del registro y leerlo. Mientras que la búsqueda secuencial es una operación O(n), el acceso directo es O(1); no importa cuan grande sea el archivo, con un solo desplazamiento es posible extraer el registro que se quiera.

El acceso directo implica saber donde esta el comienzo del registro requerido.

Si un archivo es una secuencia de registros, entonces el **NRR** (número relativo de registro) de un registro proporciona su posición relativa con respecto al principio del archivo.

Si los registros son de longitud fija, entonces se puede usar el NRR de un registro para calcular la distancia en bytes del inicio del registro en relación con el inicio del archivo. Dado un archivo de longitud fija de tamaño r, la distancia en bytes de un registro con un NRR de n es n x r.

Cuando se usan registros de longitud variable, el NRR indica la posición relativa del registro que se quiere pero se tiene que leer secuencialmente en el archivo, contando los registros que se llevan, para tener el registro que se quiere.

Registros de encabezado

Un **registro de encabezado** es un registro colocado al inicio de un archivo, que se usa para guardar información acerca de su contenido y organización.

El registro de encabezado tiene una estructura diferente de la que tienen los registros de datos del archivo. La diferencia entre los registros de encabezado y los de datos se acentúa al incluir información como la longitud de los registros de datos, la fecha y la hora de la actualización más reciente del archivo, etc.

Archivos serie y secuenciales

Un **archivo serie** es un archivo donde cada registro es accesible solo luego de procesar su antecesor. Estos archivos son simples de acceder.

Un **archivo secuencial** es un archivo donde los registros son accesibles en orden de alguna clave.

Unidad 11

Mantenimiento de archivos y eliminación de registros

La organización original de un archivo influye en como puede ser alterado, y también como puede responder al deterioro. Un **archivo volátil**, es decir, que este sometido a muchos cambios, puede deteriorarse muy rápido, a menos que se tomen medidas para ajustar su organización a los cambios.

En general, las modificaciones pueden ser una de estas tres formas: agregar un registro, actualizar un registro y eliminar un registro.

Si la única clase de cambios posible en un archivo es agregar registros, no suceden deterioros. Solo cuando se actualizan registros de longitud variable, o cuando se eliminan registros de longitud fija o variable es que el mantenimiento se vuelve complejo. La actualización puede verse como una eliminación seguida de una inserción.

Compactación del almacenamiento

Cualquier estrategia de eliminación de registros debe proporcionar alguna forma de reconocer los registros que se han eliminado. Un enfoque sencillo y que normalmente funciona, es colocar una marca especial en cada registro eliminado.

Una vez que se puede reconocer un registro eliminado, la siguiente cuestión es como reutilizar el espacio del registro. La **compactación de almacenamiento** no hace nada para reutilizar de inmediato el espacio. Los registros se marcan simplemente como eliminados y se dejan en el archivo durante un lapso.

Un efecto colateral positivo es que normalmente permite al usuario anular la eliminación de un registro con muy poco esfuerzo.

El espacio de todos los registros eliminados se recupera al mismo tiempo. Después de que los registros eliminados se han acumulado por algún lapso, se emplea un programa especial de compactación de almacenamiento que reconstruye el archivo ya sin los registros eliminados. Esto puede utilizarse con registros de longitud fija y variable.

La decisión respecto a la frecuencia con que se ejecuta el programa de compactación de almacenamiento puede basarse en el número de registros eliminados o en el calendario.

Eliminación de registros de longitud fija

Para reutilizar el espacio liberado, es necesario saber que registros están eliminados y donde están. Si se trabaja con registros de longitud fija y se está dispuesto a hacer una búsqueda secuencial en el archivo antes de agregar un registro, se podrá proporcionar esta garantía.

Sin embargo, este proceso es muy lento cuando el programa es interactivo y el usuario se ve obligado a esperar que ocurra la inserción del registro. Para que esto se realice más rápido se necesita:

- Una forma de saber de inmediato si hay lugares vacíos en archivo.
- Una forma de saltar directamente a uno de esos lugares, en caso de existir.

Una **lista ligada** es una estructura de datos en la que cada elemento o nodo contiene algún tipo de referencia sobre su sucesor en la lista.

Cuando una lista está compuesta de registros eliminados que se han convertido en espacio disponible dentro del archivo, se llama **lista de disponibles**. Al insertar un registro nuevo en un archivo de registros de longitud fija, cualquier registro disponible es bueno.

El modo más sencillo de manejar una lista es como si fuera una **pila**. Una pila es una lista en donde todas las inserciones y eliminaciones de nodos se realizan en uno de los dos extremos. Si el apuntador del tope de la pila es fin de lista, no hay espacio para reutilizar. Por lo tanto, se inserta al final. Si el apuntador es válido, hay un espacio y apunta al lugar donde hay que insertar.

Eliminación de registros de longitud variable

Se necesita una estructura de archivo donde se defina la longitud de cada registro, colocando un contador de bytes del contenido del registro al inicio de cada uno.

Como esta estructura no considera el número de campos de un registro, se puede manejar el contenido de un registro de longitud variable, como se hizo con los de longitud fija. Como no se puede calcular la distancia en bytes para los registros de longitud variable a partir de sus NRR, son las ligas las que deben contenerla.

Se deben cumplir dos condiciones para reutilizar un registro:

- El registro debe haber sido eliminado.
- El registro debe tener el tamaño correcto.

Como se necesita buscar, no se puede organizar la lista de disponibles como una pila.

Fragmentación del almacenamiento

El espacio desperdiciado dentro de un registro se llama **fragmentación interna**. Cuando se trabaja con registros de longitud fija, se procura minimizar la fragmentación interna eligiendo una longitud que sea lo más aproximada posible a la que se necesita para cada registro.

Cuando se trabaja con registros de longitud variable, la fragmentación se da cuando se elimina un registro y se reemplaza por otro más corto. En este caso se puede dividir el espacio en dos partes: una para guardar el nuevo registro y la otra para colocar de nuevo en la lista de disponible. En este caso puede haber **fragmentación externa**, ya que el espacio está en realidad en la lista de disponibles, y no encerrado en algún otro registro, pero está demasiado fragmentado como para reutilizarse.

Una de las formas de eliminar la fragmentación externa es la compactación del almacenamiento. Otros dos enfoques son:

- Cuando es posible, se combinan fragmentos pequeños para crear fragmentos más grandes y útiles. Si el programa advierte que dos entradas de la lista de disponibles son adyacentes, puede combinarlas para hacer una entrada mayor. Esto se llama unir los huecos en el espacio de almacenamiento.
- Intentar minimizar la fragmentación antes de que suceda, adoptando una estrategia de colocación que el programa pueda usar cuando selecciona una entrada de la lista de disponibles.

Estrategias de colocación

Una estrategia de colocación es un mecanismo de selección del espacio en la lista de disponibles, a fin de usarlo para almacenar un nuevo registro que se agregue al archivo.

Primer ajuste: selecciona la primera entrada disponible que sea lo suficientemente grande para almacenar el nuevo registro. Es una estrategia sencilla.

Mejor ajuste: encuentra la entrada disponible cuyo tamaño se aproxime más al que se necesita para almacenar el nuevo registro. Esta estrategia genera fragmentación interna.

Peor ajuste: selecciona la entrada más grande, sin importar cuan pequeño sea el nuevo registro. Como esto deja la mayor entrada posible para reutilización, en algunos casos, el peor ajuste ayuda a minimizar la fragmentación externa.

Unidad 12

Búsqueda de información en un archivo

Hasta ahora la única forma de extraer o encontrar un registro con algún grado de rapidez es buscar por su número relativo de registro (NRR). Si el archivo tiene registros de longitud fija, conocer el NRR permite saltar al registro usando acceso directo.

¿Qué pasa si no se conoce el NRR del registro que se desea? Es mucho más factible que se conozca la identidad de un registro por su llave.

El acceso por llave implica una búsqueda secuencial, examinando un registro tras otro hasta que se encuentra el que tiene la llave. En el caso de que no exista el registro con dicha llave o que se sospeche que puede haber más de un registro con la misma llave, se debe revisar todo el archivo.

Hay muchas formas mejores que la secuencial para manejar los accesos mediante llave.

Búsqueda binaria

Se compara la llave de búsqueda con la llave que está a la mitad del archivo. El resultado de la comparación, indica cuál mitad del archivo contiene el registro buscado. Se vuelve a comparar la llave con la llave de la mitad, para saber en cual cuarto del archivo está el registro. Este proceso se repite hasta que se encuentre el registro buscado o se haya reducido el número de registros a cero.

Esta clase de búsqueda, donde la mitad de los objetos de información restantes se eliminan con cada comparación, se llama búsqueda binaria.

Una búsqueda binaria en un archivo con n registros realiza a lo sumo $\log_2 n + 1$ comparaciones, y en promedio $\log_2 (n) + 1/2$ comparaciones. Por lo tanto, se dice que una búsqueda binaria es $O(\log_2 n)$. En una búsqueda secuencial se requieren a lo sumo n comparaciones, y en promedio 1/2 n, lo cual quiere decir que una búsqueda secuencial es O(n).

La diferencia entre una búsqueda secuencial y una binaria se hace más notoria a medida que se incrementa el tamaño del archivo en donde se busca. Si se duplica el tamaño del archivo, se duplica el número de comparaciones necesarias para la búsqueda secuencial, en cambio, cuando se usa la búsqueda binaria, duplicar el tamaño del archivo implica solo hacer una conjetura mas, para el peor caso.

El problema de la búsqueda binaria es que funciona solo cuando el archivo está ordenado en términos de la llave que se está usando en la búsqueda. De este modo, para hacer uso de la búsqueda binaria se debe poder clasificar el archivo con base en una llave.

Clasificación de un archivo de disco en memoria RAM

Cualquier algoritmo de clasificación interna requiere pasar varias veces por la lista que se clasifica, comparando y reorganizando los elementos. Algunos objetos de información de la lista se transportan grandes distancias desde su posición inicial. Si este algoritmo fuese aplicado en forma directa a los datos almacenados en un disco, habría bastantes saltos, desplazamientos y relectura de datos. Esto sería una operación muy lenta.

Si el contenido completo del archivo puede almacenarse en memoria RAM, una alternativa es trasladar el archivo completo del disco a la memoria, y después clasificarlo ahí.

El primer paso consiste en trasladar todos los registros del archivo a la memoria. Se supone que se trata de un archivo de registros de longitud fija y que tiene un registro de encabezado en el cual se encuentra la cantidad de registros que hay en el archivo. Una vez que se han leído los registros, se tiene un arreglo de vectores de caracteres del mismo tamaño, y cada renglón del arreglo esta compuesto del contenido de un registro. A este arreglo lo llamamos REGISTROS[].

Como la clasificación se debe hacer sobre la forma canónica de la llave, se necesita extraer un segundo arreglo que contenga solo las llaves en forma canónica y después hacer la clasificación sobre esas llaves. Además este arreglo debe tener un segundo campo que contiene el NRR del registro asociado con la llave para poder relacionar las llaves con los registros de donde fueron extraídas. Este arreglo se llamara NODOSLLAVE[]. Este mecanismo de construcción de una cadena de referencia se conoce como **indirección**.

Cada elemento de NODOSLLAVE[] contiene una cadena de caracteres. Si se clasifica el arreglo de llaves moviendo físicamente los elementos, nodo llave, habría que hacer muchas copias de cadenas. La clasificación puede requerir un gran número de movimientos.

Cada llave es un elemento del arreglo NODOSLLAVE[] y, por lo tanto, puede hacer referencia a ella a través de un subíndice entero. Se pueden reordenar las llaves sin tener que moverlas si se forma un tercer arreglo, compuesto simplemente por los subíndices del arreglo NODOSLLAVE[]. A este arreglo lo llamamos INDICE[].

Con el arreglo INDICE[] se puede ordenar el arreglo NODOSLLAVE[] reubicando los valores enteros dentro de INDICE[]. Además, como no se necesita mover los elementos del arreglo NODOSLLAVE[], tampoco es necesaria la referencia de los NRR dentro de NODOSLLAVE[].

Una vez que los elementos de INDICE[] están ordenados se puede escribir todos los REGISTROS[] clasificados de acuerdo al orden de la llave.

Una clasificación en memoria RAM funciona solo con archivos pequeños.

Clasificación por llave (clasifllave)

La diferencia con la clasificación en memoria RAM (clasifram) es que no se leen los registros reales en memoria, se leen solo las llaves canónicas.

Clasifllave mantiene los mismos componentes que clasifram. El proceso de clasificación reacomoda el arreglo de subíndices de las llaves canónicas, exactamente como el clasifram. Pero como clasifllave nunca pone todo el conjunto de registros en la memoria, no necesita usar tanta como el clasifram. Esto significa que clasifllave puede clasificar archivos más grandes, dada la misma cantidad de memoria RAM.

Las diferencias son:

- En lugar de leer todos los registros en un arreglo en memoria RAM, se lee cada registro en un BUFFER temporal y luego se desecha.
- Cuando se escriben los registros ordenados tienen que leerse por segunda vez, porque no se tienen todos almacenados en memoria RAM.

Cuando se leen los registros antes de transcribirlos al nuevo archivo, no se está leyendo en forma secuencial, sino que se trabaja en el orden de clasificación. Dado que hay que desplazarse a cada registro y leerlo antes de escribirlo, la creación del archivo clasificado requiere tantos desplazamientos aleatorios dentro del archivo de entrada como registros existen.

¿Por qué molestarse en escribir de nuevo el archivo?

En vez de crear una copia nueva y clasificada del archivo para la búsqueda, se crea un segundo tipo de archivo, un archivo de índices, que se usa en conjunción con el archivo original. Si se está buscando un registro en particular, se hace la búsqueda binaria en el archivo de índices y después se usa el NRR almacenado en el registro del archivo de índices, para encontrar el registro correspondiente del archivo original.

Como los programas de clasificación, las distintas estructuras de índices implican extraer llaves de los registros de un archivo. Estas estructuras de índices por lo regular también implican indirección que es el acceso a los registros por medio de una cadena de uno o más apuntadores, o referencias a números relativos de registros. Como en la clasificación por llave, se centran en mantener y organizar un archivo de índices, en lugar de reorganizar simplemente los registros originales.

Registros fijos

Se dice que un registro está fijo cuando existen otros registros o estructuras de archivos referidas a él mediante su posición física. Está fijo en el sentido de que no se tiene la libertad de alterar la posición física del registro, ya que al hacerlo se destruiría la validez de las referencias físicas al registro. Estas referencias se convierten en apuntadores suspendidos inútiles.

Indice

Un índice es una herramienta para encontrar registros en un archivo. Consiste en un campo llave mediante el cual se busca el índice, y un campo de referencia que indica donde encontrar el registro del archivo de datos asociado con una llave en particular.

Como trabaja por indirección, un índice permite imponer un orden en un archivo sin que realmente se reacomode. Así no solo se evita desacomodar los registros fijos, sino que también los costos de aspectos tales como la adición de registros son mucho menores que en un archivo clasificado.

La indización también proporciona varios caminos de acceso a un archivo y acceso por llave a archivos de registros de longitud variable.

Hay dos tipos de índices que pueden usarse:

- Indice denso: aparece un registro índice para cada valor de la clave de búsqueda en el archivo. El registro contiene el valor de la clave de búsqueda y un puntero al registro.
- Indice escaso: se crean registros índices solamente para algunos de los registros. Para localizar un registro, encontramos el registro índice con valor de la clave de búsqueda que estamos buscando. Empezamos en el registro al que apunta el registro índice y seguimos los punteros del archivo hasta encontrar el registro deseado.

Existen dos técnicas de indización:

- Archivos secuenciales indizados: consiste en un archivo de datos de entrada secuencial, divididos en bloques de registros, junto con un índice escaso.
- Archivos no secuenciales: consiste en un archivo de datos de entrada no secuencial, divididos en bloques de registros junto con un índice denso.

Archivos secuenciales indizados

La estructura del archivo de índices es muy simple, es un archivo de registros de longitud fija, donde cada registro tiene dos campos de longitud fija: un campo llave y un campo de distancia en bytes. Para cada registro del archivo de datos hay un registro en el archivo de índices.

El índice está clasificado, a diferencia del archivo de datos.

Se trabaja con dos archivos, el archivo de datos y el archivo de índices. El archivo de índices es más fácil de manejar que el de datos porque emplea registros de longitud fija, y porque suele ser mucho más pequeño que el archivo de datos.

Al requerir que el archivo de índices tenga registros de longitud fija, se impone un límite en los tamaños de las llaves. El empleo de un campo llave pequeño y fijo en el índice puede causar problemas si la identidad única de una llave se pierde cuando se coloca en el campo fijo de índice.

Operaciones básicas en un archivo indizado

Indice en memoria

El proceso de mantener los archivos clasificados para permitir búsqueda binaria de registros puede ser muy costoso. Una de las grandes ventajas de un índice simple con un archivo de datos con entrada secuencial es que la adición de registros es mucho más rápida que con un archivo de datos clasificado, siempre y cuando el índice sea lo suficientemente pequeño para almacenarse completo en la memoria.

Mantener el índice en la memoria también permite encontrar registros por llave más rápido con un archivo indizado que con uno clasificado, porque la búsqueda binaria puede efectuarse por completo en la memoria. Una vez que se encuentra la distancia en bytes del registro de datos, todo lo que se requiere es un desplazamiento para recuperar el registro.

Crear los archivos vacíos originales

Tanto el archivo de índice como el de datos se crean como archivos vacíos, con registros de encabezado y nada más. Esto puede llevarse a cabo creando los archivos y escribiendo los encabezados en ambos archivos.

Carga del índice en la memoria

Se define un arreglo INDICE[] para almacenar los registros de índice. Cada elemento del arreglo tiene la estructura de un registro de índice. La carga del archivo de índices en la memoria es solo cuestión de leer y guardar el registro de encabezado del índice, y después leer los registros del archivo de índice en el arreglo INDICE[].

Reescritura del archivo de índices de la memoria

Cuando se termina el procesamiento de un archivo indizado es necesario reescribir INDICE[] de vuelta en el archivo de índices, si el arreglo ha cambiado.

Es importante considerar lo que sucede si esta reescritura del índice no se realiza o se efectúa en forma incompleta. Un programa debe contener al menos las siguientes dos defensas:

- Debe existir un mecanismo que permita al programa saber cuando el índice no está actualizado. Una posibilidad consiste en activar una bandera de estado cuando la copia del índice en la memoria cambie. Esta bandera se puede escribir en el registro de encabezado del archivo de índices del disco en cuanto se haya leído el índice en la memoria, y se puede desactivar cuando el índice se haya reescrito.
- Si un programa detecta que un índice no está actualizado, debe acceder a un procedimiento que reconstruya el índice a partir del archivo de datos.

Adición de registros

Agregar un registro nuevo al archivo de datos requiere que también se agregue un registro al archivo de índices. Cuando se agrega un registro de datos se debe conocer la distancia en bytes inicial de la posición física del archivo donde se escribió el registro. Esta información debe colocarse en el arreglo INDICE[], junto con la forma canónica de la llave del registro.

Como el arreglo INDICE[] se mantiene en orden clasificado por llaves, la inserción de índices nuevos probablemente requerirá algún reacomodo del índice. La gran diferencia entre el trabajo requerido por los registros del índice y el requerido por un archivo clasificado es que el arreglo INDICE[] está contenido por completo en la memoria. Todo el reacomodo de los índices puede hacerse sin ejecutar ningún acceso al archivo.

Una desventaja de este procedimiento directo es el tiempo que toma recorrer los registros de los índices.

Eliminación de registros

Cuando se elimina un registro del archivo de datos, también debe eliminarse el registro correspondiente del archivo de índices. Como el índice esta contenido en un arreglo durante la ejecución del programa, la eliminación del registro de índice y el desplazamiento de los demás registros para agrupar el espacio puede ser una operación no muy costosa. Esto se puede hacer aun más rápido usando alguna indirección adicional. Una alternativa es marcar como eliminado el registro de índice, igual que puede marcarse el registro de datos correspondiente.

Actualización de registros

La actualización cambia el valor del campo de llave: puede traer consigo un reacomodo del archivo de índices, así como del de datos. La forma más fácil de concebir esta clase de cambio es como una eliminación seguida de una adición.

La actualización no afecta el campo de la llave: no requiere reacomodo del archivo de índices, pero puede implicar el reacomodo del archivo de datos. Si el tamaño del registro no cambia, o disminuye por la actualización, el registro puede escribirse directamente en el espacio que tenía, pero si aumenta por la actualización, se tendrá que encontrar una nueva entrada para el registro. En este caso, se debe modificar la dirección contenida en el registro índice.

Indices demasiado grandes para almacenarse en memoria

Si el índice es demasiado grande como para almacenarlo en memoria, entonces el acceso y el mantenimiento debe hacerse en el almacenamiento secundario. Con índices simples, el acceso al disco tiene las siguientes desventajas:

- La búsqueda binaria del índice requiere varios desplazamientos.
- El reacomodo de índices debido a la adición o eliminación de registros requiere mover o clasificar registros en el almacenamiento secundario.

Cuando un índice simple es demasiado grande para almacenarse en memoria, debe considerarse el uso de:

- Una organización por dispersión, cuando la velocidad de acceso tiene máxima prioridad.
- Un índice estructurado en forma de árbol, como un árbol B, cuando se necesita flexibilidad tanto en el acceso por llave como en el acceso ordenado y secuencial.

El uso de índices simples en el almacenamiento secundario, proporciona algunas ventajas importantes que no tiene el uso de archivos clasificados por llave, aun si el índice no puede almacenarse en memoria:

- Un índice simple hace posible la búsqueda binaria para obtener acceso por llave a un registro en un archivo de registros de longitud variable.
- Si los registros de índices son considerablemente más pequeños que los registros de datos del archivo, la clasificación y el mantenimiento del índice pueden ser menos costosos de lo que sería la clasificación y el mantenimiento del archivo de datos.
- Si hay registros fijos en el archivo de datos, el uso de un índice permite reacomodar las llaves sin tener que mover los registros de datos.
- Pueden emplearse varios índices para proporcionar diversas formas de ver un archivo de datos.

Indices secundarios

Los punteros en el índice secundario no señalan directamente al archivo. En vez de ello, cada uno de esos punteros señala a una cubeta que contiene punteros al archivo.

Un índice secundario puede contener llaves duplicadas. Permite almacenar juntos todos los punteros de un valor de clave de búsqueda secundaria determinado. Un enfoque así, es útil en ciertos tipos de consultas para los que podemos hacer una parte considerable de procesamiento utilizando únicamente los punteros. Para las claves primarias, podemos obtener todos los punteros para un valor de la clave de búsqueda primaria determinada utilizando una revisión secuencial.

El índice secundario puede ser denso o escaso. Si es denso, el puntero de cada cubeta individual señala a los registros con el valor de la clave de búsqueda apropiado. Si el índice secundario es escaso, el puntero de cada cubeta individual señala a los registros con valores de la clave de búsqueda en el rango apropiado. En este caso cada entrada de cubeta es un puntero único o un registro con dos campos: un valor de la clave de búsqueda y un puntero a algún registro del archivo.

Asociando un valor de la clave de búsqueda con cada puntero de la cubeta, eliminamos la necesidad de leer registros con un valor de la clave de búsqueda secundaria distinto del que estamos buscando.

La estructura de la cubeta puede eliminarse si el índice secundario es denso y los valores de la clave de búsqueda forman una clave primaria.

Los índices secundarios mejoran el rendimiento de las consultas que utilizan las claves que no son primarias. Sin embargo, implican un gasto extra considerable en la modificación de la base de datos.

Operaciones:

- Adición de registros: el costo que esto implica es similar al que representa agregar un registro al índice primario.
- Eliminación de registros: por lo general, la eliminación de un registro implica la eliminación de todas las referencias a ese registro en el sistema de archivos, de modo que eliminar un registro del archivo de datos significa la eliminación no solo del registro correspondiente en el índice primario, sino también de todos los registros de los índices secundarios que hacen referencia a este registro del índice primario. El problema es que la eliminación de un registro puede implicar el reacomodo de los registros restantes.
- Actualización:
  - ✓ La actualización cambia la llave secundaria: si se cambia la llave secundaria, probablemente habrá que reacomodar el índice secundario de tal forma que permanezca en el orden de clasificación. Esto puede ser costoso.

- ✓ La actualización cambia la llave primaria: tiene gran repercusión en el índice de la llave primaria, pero con frecuencia solo requiere que se actualice el campo de referencia afectado en todos los índices secundarios.
- ✓ La actualización se restringe a los otros campos: las actualizaciones que no afectan los campos de la llave primaria o secundaria no afectan el índice de las llaves secundarias, aunque la actualización sea considerable.

Listas invertidas

Las estructuras de índices secundarios que se desarrollaron hasta ahora ocasionan dos problemas:

- El archivo de índices tiene que reacomodarse cada vez que se agrega un registro nuevo al archivo.
- Si hay llaves secundarias duplicadas, el campo de llave secundaria se repite para cada entrada. Esto desperdicia espacio, ya que los archivos se agrandan más de lo necesario. Los archivos de índices grandes tienen menos posibilidades de entrar en la memoria auxiliar.

Una solución a estos problemas es cambiar la estructura del índice secundario, de tal forma que se asocie un **arreglo de referencias** con cada llave secundaria. Por ejemplo, puede usarse una estructura de registro que permita asociar hasta cuatro campos de referencia con una llave secundaria.

Si se agrega un registro, solo necesita modificarse el registro correspondiente en el índice secundario insertando un campo de referencia. Como no se agrega otro registro al índice secundario, no hay necesidad de reacomodar nada.

Esta estructura solo proporciona espacio para cuatro referencias asociadas a una llave. En el caso de que se haya mas de cuatro referencias asociadas a una llave, se necesitara un mecanismo que mantenga la información de las referencias adicionales.

Aunque la estructura ayuda a evitar el desperdicio de espacio ocasionado por la repetición de llaves idénticas, estos ahorros de espacio tienen un costo alto. Al ampliar la longitud fija de cada uno de los registros de índice secundario para almacenar más campos de referencias, se puede perder más espacio por fragmentación interna que el que se ganó por no repetir las llaves idénticas.

Otra solución: ligar la lista de referencias

A los archivos que son como los índices secundarios, en los que una llave secundaria lleva a un conjunto de una o más llaves primarias, se les llama **listas invertidas**. El sentido en el que se invierte una lista debe quedar claro si se considera que se trabaja retrocediendo de una llave secundaria a la llave primaria y al registro mismo.

Como el nombre de la lista invertida indica, se trata de una lista de referencias de llaves primarias.

Una situación ideal sería hacer que cada llave secundaria apuntara a una lista diferente de referencias de llaves primarias. Cada una de estas listas podría crecer hasta tener la longitud que se necesite.

A diferencia de la estructura de registros que asigna un espacio fijo de referencias, las listas podrían contener cientos de referencias, si fuera necesario y seguir requiriendo solo una instancia de llave secundaria. Por otra parte, si una lista requiere solo un elemento, entonces no se pierde espacio por fragmentación interna.

Lo más importante es que solo se necesita reacomodar el archivo de llaves secundarias cuando se añade un índice al archivo.

¿Cómo puede establecerse un número ilimitado de listas diferentes, cada una de longitud variable, sin crear un gran número de archivos pequeños? Podría redefinirse el índice secundario de modo que conste de registros con dos campos: un campo de llave secundaria y un campo que contenga el número relativo de registro de la primera referencia a la llave primaria correspondiente en la lista invertida. Las referencias reales de las llaves primarias asociadas con cada llave secundaria podrían almacenarse en un archivo secuencial separado. El último registro de la lista contiene como número relativo de registro el valor -1.

Asociar el archivo de índice secundario a un nuevo archivo que contenga listas ligadas de referencias tiene algunas ventajas:

- El único momento en el que se requiere reacomodar el archivo de índices secundarios es cuando se añade o cambia un registro del archivo de índices. Borrar o añadir una referencia en la lista solo implica cambiar el archivo de listas.
- En el caso de que sea necesario reacomodar el archivo de índices secundarios, la tarea es más rápida, ya que existen menos registros y cada registro es más pequeño.

- Dado que hay menos necesidad de clasificaciones, es menor el precio que hay que pagar por mantener los archivos de índices secundario en el almacenamiento secundario, por lo que queda más lugar en memoria RAM para otras estructuras.
- El archivo de listas es de entradas secuenciales, por lo cual, nunca necesitará clasificarse.
- Como el archivo de listas es de registros de longitud fija, sería muy fácil desarrollar un mecanismo para reutilizar el espacio de los registros eliminados.

Una desventaja para este tipo de organizaciones de archivos es que no se garantiza la localidad. La **localidad** existe en un archivo cuando los registros a los cuales se accederá en una secuencia temporal dada, se encuentran en la proximidad física uno con otro en el disco. En una estructura ligada y secuencial como esta, es poco probable que haya localidad en los agrupamientos lógicos de los campos de referencia para una determinada llave secundaria. Esta falta de localidad significa que recuperar las referencias de una llave que tenga una lista larga de referencias implicaría una gran cantidad de desplazamientos en el disco.

Una solución para este problema de desplazamiento es mantener el archivo de listas en la memoria.

Indices selectivos

Un índice selectivo contiene llaves solo para una porción de los registros del archivo de datos. Dicho índice permite al usuario ver un subconjunto específico de los registros del archivo.

La información de índices selectivos podría combinarse en operaciones de conjunción booleana para responder a distintas solicitudes. Algunas veces, los índices selectivos son útiles cuando el contenido del archivo de datos entra, de manera natural y lógica, en varias categorías generales.

Enlaces

¿En qué momento se liga la llave a la dirección física de su registro asociado?

El enlace que se realiza en el momento de la construcción del archivo permite un acceso más rápido. Una vez que se ha encontrado el registro de índice correcto, se tiene la distancia en bytes del registro de datos que se está buscando. La mejora en el desempeño es particularmente notable cuando los archivos de llaves primarias y secundarias se usan en el almacenamiento secundario y no en la memoria.

La desventaja del enlace directo en el archivo, el enlace fuertemente acoplado, es que las reorganizaciones del archivo de datos deben provocar modificaciones en todos los archivos de índices enlazados.

Si se posterga el enlace hasta el tiempo de ejecución, cuando los registros se usan realmente, puede desarrollarse un sistema de llaves secundarias que implique una cantidad mínima de reorganización cuando se agregan o eliminan registros.

Otra ventaja importante es que asociar las llaves secundarias con los campos de referencia que consisten en llaves primarias permite que el índice de llaves primarias actúe como una especie de revisión final, que asegura que el registro esté en realidad en el archivo.

Con un esquema de enlace al momento de la extracción de información, se necesita recordar hacer el cambio en un solo lugar, el índice de las llaves primarias, en caso de mover un registro de datos. Con un sistema de enlace fuertemente acoplado, hay muchos cambios que tienen que realizarse en forma exitosa para que el sistema permanezca consistente internamente.

Procesos secuenciales coordinados

Las operaciones secuenciales coordinadas implican el procesamiento coordinado de dos o más listas secuenciales para producir una única lista de salida. A veces el procesamiento produce una intercalación o unión de las listas de entrada, otras veces el objetivo es una correspondencia o una intersección de las listas, y otras la operación es una combinación de correspondencia e intercalación.

La correspondencia es el proceso de formar un archivo de salida clasificado compuesto por todos los elementos comunes en dos o más archivos de entrada clasificados. Hay varios aspectos que hay que tener en cuenta para que funcione bien:

- Asignación de valores iniciales.
- Sincronización.
- Manejo del estado de fin de archivo.
- Reconocimiento de errores.

La intercalación es el proceso de formar un archivo de salida clasificado compuesto por la unión de los elementos de dos o más archivos de entrada clasificados.

Intercalación de k formas

Se pretende intercalar k listas de entrada para crear una sola lista de salida ordenada secuencialmente.

Se toma el primer elemento de cada lista y se lleva a la salida el mínimo de estos. Después se avanza hacia delante en la lista de la cual se tomo el elemento. Este proceso se continúa hasta que todas las listas estén vacías.

En el caso de registros duplicados, el movimiento es hacia delante en cada lista.

La intercalación de k formas funciona bien cuando k no es mayor de 8, aproximadamente. Cuando se comienza a intercalar un mayor número de listas, el número de comparaciones secuenciales para encontrar la llave de valor mínimo comienza a ser costoso. Si hubiera necesidad de intercalar mucho más de ocho listas, se reemplazaría el ciclo de comparaciones por un árbol de selección.

Intercalación como forma de clasificación de archivos grandes en disco

Se crea un subconjunto clasificado del archivo completo, transfiriendo los registros a memoria RAM hasta que el área de trabajo este casi llena, clasificando los registros en esta área de trabajo, y luego transcribiendo los registros clasificados al disco como un subarchivo clasificado. A estos subarchivos clasificados se los llama porciones.

Una vez que se tienen las k porciones en k archivos separados en disco, puede efectuarse una intercalación de k formas, para crear un archivo completamente clasificado que contenga todos los registros originales.

Esta solución al problema de la clasificación tiene las siguientes características:

- Puede clasificar archivos grandes.
- La lectura del archivo de entrada durante la creación de la porción es completamente secuencial, y por lo tanto es más rápida que en la entrada que requiere desplazamientos.
- La lectura de cada porción durante la intercalación y la escritura de los registros clasificados también es secuencial.
- El proceso se retarda debido a los desplazamientos en el disco durante la fase de intercalación.
- Para realizar la intercalación de n porciones, se requieren n accesos a cada porción, por lo tanto, necesitamos n^2 accesos.

Hay dos formas de reducir el número de desplazamientos requeridos para el paso de intercalación de la clasificación:

- Efectuar la intercalación en más de un paso, reduciendo el orden de cada intercalación e incrementando el tamaño del buffer.
- Incrementar el tamaño de las porciones iniciales clasificadas.

Intercalación en varios pasos

Supongamos que tenemos 40 particiones, en vez de intercalar las 40 particiones a la vez, se podrían intercalar como cinco conjuntos de ocho porciones cada uno, seguidos de una intercalación de cinco formas de las porciones intermedias.

Este método requiere que se lea dos veces cada registro: una para formar las porciones intermedias, y otra, para formar el archivo clasificado final.

Casi siempre resulta ventajoso pagar el precio de leer los registros más de una vez para reducir el número necesario de desplazamientos en disco.

Incremento de las longitudes de las porciones mediante el uso de selección por reemplazo

Una partición inicial más grande implica una intercalación de menor orden con buffers más grandes y un menor número de desplazamientos.

Este método utiliza la selección por reemplazo. La **selección por reemplazo** se basa en la idea de seleccionar siempre de la memoria el registro cuya llave tenga el valor más bajo, enviar a la salida ese registro y después reemplazarlo en memoria con un nuevo registro de la lista de entrada. Cuando se

obtienen registros nuevos cuyas llaves son mayores que las de los registros enviados recientemente a la salida, finalmente forman parte de la porción que se está creando. Cuando los registros nuevos tienen llaves menores que las de los enviados más recientemente a la salida, se guardan para la siguiente porción.

La selección por reemplazo produce por lo general porciones considerablemente más grandes que las que pueden crearse mediante clasificaciones en memoria RAM, y por tanto pueden ayudar a mejorar el desempeño en la clasificación por intercalación. Sin embargo, al usar la selección por reemplazo con clasificaciones e intercalaciones en disco, se debe cuidar que los desplazamientos adicionales requeridos no eliminen los beneficios que aportan el tener porciones más grandes por intercalar.

En la selección por reemplazo los registros se deben leer de a uno, esto es imposible, se utilizan buffers que ocupan parte de la RAM. La solución es la selección natural.

En la **selección natural** los registros que quedan dormidos se pasan a un buffer secundario, y permiten que entren nuevos elementos del archivo de entrada, cuando el buffer secundario se llena se termina de dormir elementos.

En la selección natural las particiones quedan con más elementos y aprovecha el espacio de memoria, necesitamos algún buffer más pero ahorra memoria.

Intercalación balanceada

Usa el mismo número de dispositivos de entrada que de salida. Tiene dos conjuntos de buffers, el de entrada y el de salida.

Método:

1. Cada partición (con M elementos) se acomoda en un archivo de entrada.
2. Se realiza un merge entre los archivos de entrada produciendo archivos de salida con M^2 registros ordenados.
3. Continuar 2 hasta terminar los archivos de entrada.
4. Convertir los archivos de entrada en archivos de salida y viceversa.
5. Repetir 2 hasta formar una partición con todos los elementos ordenados.

Intercalación en varias fases

La distribución de las porciones es tal que al menos la intercalación inicial es de J-1 formas (j es el número de buffers de entrada), y donde la distribución de las porciones en los buffers es tal que la intercalación funciona eficientemente en cada paso.

Unidad 13

Arboles binarios

Estructuras de datos donde cada nodo tiene dos sucesores, a izquierda y a derecha.

Una ventaja es que si se agrega un nuevo dato al archivo, sólo se necesita ligarla con el nodo hoja apropiado para crear un árbol en el que la eficiencia de la búsqueda es tan buena como la que se tiene con una búsqueda binaria en una lista clasificada.

Un árbol está **balanceado** cuando la altura de la trayectoria más corta hacia una hoja no difiere de la altura de la trayectoria más larga en más de un nivel.

Una desventaja de los árboles binarios es que se desbalancean fácilmente.

Arboles AVL

Un árbol **AVL** es un árbol balanceado en altura. Esto significa que existe un límite en la magnitud de la diferencia que se permite entre las alturas de cualesquiera dos subárboles que comparten una raíz común. En un árbol AVL la diferencia máxima permitida es 1. Por lo tanto, a un árbol AVL se le llama árbol-1 balanceado en altura, o árbol BA(1), y es miembro de una clase más general de árboles balanceados en altura conocidos como **árboles BA(k)**, a los cuales se permite estar k niveles fuera de balance.

Las dos características en las que radica la importancia de los árboles AVL son:

- Al establecer una diferencia máxima permitida en la altura de dos subárboles, los árboles AVL garantizan un cierto nivel mínimo de desempeño en la búsqueda.
- El mantener un árbol en forma de AVL conforme se insertan nodos nuevos implica el uso de una de las cuatro posibles rotaciones. Cada una de ellas se restringe a una única área local del árbol. Las rotaciones más complejas requieren solo cinco reasignaciones de apuntadores.

Para un árbol completamente balanceado, el peor caso de la búsqueda para encontrar una llave, considerando N llaves posibles, busca en $\log_2(N + 1)$ niveles del árbol. Para un árbol AVL, el peor caso de la búsqueda podría ser buscar en $1.44 \log_2(N + 2)$ niveles. Este resultado es interesante, dado que los procedimientos AVL garantizan que una sola reorganización requiere no más de cinco reasignaciones de apuntadores.

Arboles binarios paginados

Dividir el árbol en páginas permite búsquedas más rápidas en almacenamiento secundario, y proporciona una extracción mucho más rápida que cualquier otra de las formas de acceso por llave.

Mientras el número de desplazamientos requeridos para el peor de los casos de la búsqueda binaria en un árbol balanceado completamente lleno es $\log_2(N + 1)$ en donde N es el número de llaves del árbol, el número de desplazamientos requeridos para las versiones paginadas de un árbol balanceado completamente lleno es $\log_{k+1}(N + 1)$ donde k es el número de llaves almacenadas en una sola página.

Todo acceso a una página requiere la transmisión de una gran cantidad de datos, la mayoría de los cuales no se usan. Sin embargo, este tiempo de transmisión adicional justifica su costo debido a que ahorra muchos desplazamientos, los cuales consumen más tiempo que las transmisiones adicionales.

Es difícil combinar la idea de paginación de estructuras de árboles con la de balanceo de estos árboles por el método AVL. Esto esta asociado con el problema de seleccionar miembros de la página raíz de un árbol o subárbol cuando el árbol se construye en la forma descendente convencional.

Arboles multicaminos

Es una generalización de los árboles binarios que en vez de tener dos punteros, tiene r registros y r+1 apuntadores.

Permiten manejar árboles con menor profundidad.

Arboles B

Tienen una estructura de forma ascendente que permite tenerlo balanceado a bajo costo. La raíz emerge en vez de colocarla y luego ubicarla donde corresponda.

Tenemos n nodos y n + 1 apuntadores, n + 1 define el orden del árbol.

Los árboles B tienen las siguientes propiedades:

- Cada nodo tiene como máximo n descendientes.
- La raíz tiene como mínimo 2 descendientes.
- Un nodo que tiene k descendientes tiene k - 1 llaves.
- Cada nodo, excepto la raíz tiene de n/2 a n descendientes.
- El número mínimo de llaves de un nodo es n-1/2 y el máximo es n-1.
- Las hojas están todas a la misma altura.

El poder de los árboles B reside en que están balanceados (no hay ramas demasiado largas); tiene poca profundidad (requiere pocos desplazamientos); permiten eliminaciones e inserciones aleatorias a un costo relativamente bajo mientras se mantiene el balance, y garantizan, al menos, un 50 por ciento de utilización del almacenamiento.

Al número máximo de descendientes que puede tener una página se le llama **orden** del árbol B.

Dos nodos son **adyacentes hermanos** si tienen el mismo padre y son apuntados por punteros adyacentes en el padre.

Formato de un nodo

P_O	R_1	P_1	R_2	P_2	R_3	P_3	R_4	P_4	R_5	P_5	Nro de registros

Con $R_1 < R_2 < R_3 < R_4 < R_5$

Búsqueda de información

El procedimiento de búsqueda se llama a sí mismo recursivamente, localiza una página y después busca en ella para encontrar la llave en niveles sucesivamente más bajos hasta que la encuentra o hasta que no puede descender más, habiendo rebasado el nivel de hoja.

Si se llega a un puntero nulo, el elemento buscado no se encuentra.

En el mejor caso se realiza una sola lectura para encontrar el elemento y en el peor caso se realizan h lecturas, donde h es la altura del árbol.

Profundidad de la búsqueda en el peor caso

Se sabe que un árbol con N llaves tiene N + 1 descendientes desde su nivel hoja. A la profundidad del árbol en el nivel hoja se le llamará d. Se puede expresar la relación entre los N + 1 descendientes y el número mínimo de descendientes de un árbol de altura d como

$$N + 1 \geq 2 \times [m/2]^{d-1}$$

puesto que se sabe que el número de descendientes de cualquier árbol no puede ser menor que el número para el peor caso de un árbol de esa profundidad. Despejando d, se llega a la siguiente expresión:

$$d \leq 1 + \log_{[m/2]}((N + 1)/2)$$

Inserción de elementos

Los registros se insertan en un nodo terminal. Hay dos casos posibles:

- El registro tiene lugar en el nodo terminal (no se produce overflow): solo se hacen reacomodamientos internos en el nodo.
- El registro no tiene lugar en el nodo terminal (se produce overflow): el nodo se divide y los elementos se reparten entre los nodos, hay una promoción al nivel superior, y esta puede propagarse y generar una nueva raíz.

Sea h la altura del árbol, entonces en el mejor caso (sin overflow) se realizan h lecturas y una escritura. En el peor caso (overflow hasta la raíz) aumenta en uno el nivel, entonces se realizan h lecturas y 2h + 1 escrituras (dos por nivel más la raíz).

Eliminación

1) Si la llave que se va a eliminar no está en una hoja, intercambiarla con su sucesor inmediato, el cuál está en una hoja.
2) Eliminar la llave.
3) Si la hoja ahora contiene por lo menos el número mínimo de llaves, no se requiere ninguna acción adicional.
4) Si la hoja ahora contiene una menos de las llaves mínimas, se examinan los hermanos izquierdo y derecho.
    a) Si un hermano tiene más del número mínimo de llaves, hay que redistribuir.
    b) Si ninguno de los dos hermanos tiene más del mínimo de llaves, se concatenan las dos hojas y la llave del medio del padre a una hoja.
5) Si las hojas se concatenaron, aplicar los pasos 3-6 al padre.
6) Si se elimina la última llave de la raíz, entonces la altura del árbol decrece.

La **concatenación** es lo inverso de la división. Como ella puede propagarse hacia arriba a través del árbol B. Así como la división promueve una llave, la concatenación debe implicar el descenso de llaves, y esto a su vez, puede causar insuficiencia en la página padre.

Cuando un nodo del árbol B cae en insuficiencia (está lleno a menos del 50 %), puede ser posible trasladarle llaves desde un nodo adyacente con el mismo padre. Esto ayuda a asegurar que se mantenga la propiedad del 50 % cubierto. Cuando las llaves se **redistribuyen**, es necesario también alterar el contenido del padre. La redistribución a diferencia de la concatenación, no implica la creación o eliminación de nodos, sus efectos son locales por completo. La redistribución también puede usarse como alternativa a la división.

La redistribución difiere de la división y de la concatenación en que no se propaga. Se garantiza que tiene efectos estrictamente locales.

En el mejor caso (se borra un nodo terminal) se necesitan h lecturas y una escritura. En el peor caso (la concatenación lleva a decrementar el nivel del árbol en uno) se necesitan 2h - 1 lecturas y h + 1 escrituras.

Arboles B*

Arbol B especial en que cada nodo está lleno por lo menos en 2/3 partes. Los árboles B* tienen las siguientes propiedades:

- Cada página tiene un máximo de m descendientes.
- Cada página, excepto la raíz y las hojas, tienen al menos (2m - 1)/3 descendientes.
- La raíz tiene al menos dos descendientes (a menos que sea hoja).
- Todas las hojas aparecen en el mismo nivel.
- Una página que no sea hoja con k descendientes contiene k - 1 llaves.
- Una página hoja contiene por lo menos [(2m - 1)/3] – 1 llaves, y no más de m - 1.

La búsqueda y la eliminación de elementos se realizan igual que en los árboles B.

Inserción de elementos

Se debe contemplar el caso de la raíz donde se permite que crezca mas que los otros nodos hasta que pueda dividirse y llenar 2 nodos en sus 2/3 partes.

Es más eficiente que en los árboles B. Si un nodo se satura se puede:

- Redistribuir a derecha: si el hermano derecho no está lleno, puedo redistribuir. Si está lleno, debo dividir.
- Redistribuir a derecha o izquierda: si el hermano de la derecha está lleno y no se puede redistribuir, se busca el de la izquierda.
- Redistribuir a derecha y a izquierda: busca llenar los 3 nodos, estos tendrán una 3/4 partes llena.

Manejo de páginas en buffers: árboles B virtuales

A medida que se leen las páginas del disco en respuesta a las solicitudes del usuario, se va llenando el buffer. Entonces, cuando se solicita una página, se accede a ella desde la memoria RAM, si se puede, evitando así un acceso al disco. Si la página no está en la memoria RAM, entonces se la transfiere al buffer desde el almacenamiento secundario, reemplazando una de las páginas que estaban allí. A un árbol B que usa un buffer de memoria RAM en esta forma se le llama **árbol B virtual**.

Al proceso de acceder al disco para extraer una página que no está en el buffer se le llama interrupción por página. Existen dos causas de interrupciones por página:

- Nunca se ha usado la página.
- Ya estuvo en el buffer pero ha sido reemplazada con una página nueva.

Se puede minimizar la segunda causa por medio del manejo de buffers. La decisión crítica aparece cuando necesita transferirse una página nueva a un buffer que ya está completo: ¿qué página habrá que reemplazar?

Un método común es reemplazar la página menos recientemente usada, se conoce como reemplazo **LRU**.

Dada una cantidad mayor de espacio para buffers, puede ser posible retener no sólo la raíz, sino todas las páginas del segundo nivel de un árbol. Para decidir que páginas se deben reemplazar puede considerarse una estrategia LRU con un factor de peso que tome en cuenta la **altura de la página**. Esto reduce el número promedio de accesos al disco.

Análisis numérico del uso de buffers con una estrategia LRU simple

Número de llaves = 2400
Total de páginas = 140
Altura del árbol = 3 niveles

Páginas en el buffer	1	5	10	20
Número promedio de accesos por búsqueda	3.00	1.71	1.42	0.97

Colocación de la información asociada con la llave

Se tiene dos opciones para almacenar la información asociada con cada llave:

- Almacenarla en el árbol B junto con la llave.
- Colocarla en un archivo separado, dentro del índice se acopla la llave con un número relativo de registro, o un byte apuntador de dirección, que hace referencia a la localización de la información en ese archivo separado.

La ventaja que tiene el primer enfoque sobre el segundo es que una vez que se encuentra la llave, no se requiere más accesos a disco. Sin embargo, si la cantidad de información asociada con cada llave es relativamente grande, entonces almacenarla con la llave reduce el número de llaves que pueden colocarse en una página del árbol B. A medida que se reduce el número de llaves por página, se reduce el orden del árbol y tiende a ser de más altura, ya que hay menos descendientes de cada página. De esta forma, la ventaja del segundo método es que, suponiendo que la información asociada es de gran longitud en relación con la longitud de la llave, colocar la información asociada en otro lugar permite construir un árbol de orden más alto y, por lo tanto, tal vez de menor altura.

Archivos secuenciales indizados

Las estructuras de archivos secuenciales indizados permiten elegir entre dos formas alternativas de visualizar un archivo:

- Indizado: el archivo puede verse como un conjunto de registros indizados por llave.
- Secuencial: se puede acceder secuencialmente al archivo (con registros físicamente contiguos, sin hacer desplazamientos), devolviendo los registros en el orden de la llave.

Hasta ahora eran métodos disjuntos, se optaba por una rápida recuperación (Arbol) o una recuperación ordenada (secuencial). Se debe encontrar una solución que agrupe ambos casos.

Mantenimiento de un conjunto de secuencias

El **conjunto de secuencias** es el nivel básico de una estructura de archivo secuencial indizado y contiene todos los registros del archivo. Cuando se lee en el orden lógico, bloque tras bloque, el conjunto de secuencias lista todos los registros por el orden de la llave.

Uso de bloques

Cuando los registros se agrupan en bloques, el bloque se convierte en la unidad básica de entrada y salida: se leen y escriben bloques enteros a la vez. En consecuencia, el tamaño de los buffers que se usan en un programa permite almacenar un bloque entero. Después de leer un bloque, todos los registros de ese bloque están en memoria RAM, donde se pueden manejar o reacomodar mucho más rápido.

Cada bloque incluye campos de liga que apuntan al bloque anterior y al siguiente, estos campos son necesarios porque los bloques consecutivos no son necesariamente adyacentes físicamente.

La inserción de nuevos registros en un bloque puede hacer que el bloque se sature. El estado de saturación puede manejarse por medio de un proceso de división de bloques. Este proceso divide los registros entre dos bloques y se reacomodan las ligas de tal modo que aún sea posible moverse a lo largo del archivo en el orden de las llaves, bloque tras bloque.

La eliminación de registros puede ocasionar que un bloque esté lleno a menos de la mitad y, por lo tanto, presente insuficiencia. La insuficiencia dentro de un bloque del conjunto de secuencias puede manejarse con los mismos tipos de procesos que para un árbol B.

Desventajas del uso de bloques:

- Una vez que se hacen las inserciones, el archivo ocupa más espacio que un archivo que no esté en bloques de registros clasificados, debido a la fragmentación interna dentro de un bloque.
- El orden de los registros no es necesariamente secuencial en forma física a lo largo del archivo. La máxima extensión garantizada de una secuencia física está contenida en un bloque.

El tamaño del bloque debe ser tal que:

- Puedan almacenarse varios bloques en memoria RAM a la vez. Por ejemplo para la redistribución.
- Se pueda acceder a uno sin necesidad de pagar el costo de un desplazamiento en el disco dentro de la operación de lectura o escritura del bloque.

Adición de un índice simple al conjunto de secuencias

Se quiere encontrar una forma eficiente de localizar algunos bloques específicos que contengan un registro en particular, conociendo la llave del registro.

Puede considerarse que cada uno de los bloques contiene un intervalo de registros. Esto ayuda a elegir el bloque que quizá tenga el registro que se busca.

La combinación de este tipo de índices con el conjunto de secuencias de bloques proporciona acceso secuencial indizado completo. Si es necesario extraer un registro específico, se consulta el índice y después se extrae el bloque correcto; si se necesita acceso secuencial, se comienza en el primer bloque y se lee en la lista ligada de bloques hasta que se han leído todos.

Este método es bueno siempre y cuando el índice pueda almacenarse en RAM.

Contenido del índice: Separadores en lugar de llaves

El propósito de un índice es asistir al usuario cuando busca un registro con una llave específica. El índice debe guiar hacia el bloque del conjunto de secuencias que contiene el registro, si es que existe en ese conjunto. El índice sirve como una especie de mapa para el conjunto de secuencias. Interesa el contenido del índice solo en tanto que ayuda a obtener el correcto en el conjunto de secuencias; el conjunto índice por si mismo no contiene respuestas, contiene solo información acerca de a donde ir para obtener respuestas.

Con esta consideración sobre el conjunto índice como un mapa puede darse el paso de reconocer que no es necesario tener llaves reales en el conjunto índice. La necesidad real es de separadores. Los separadores derivan de las llaves de los registros que limitan un bloque en el conjunto de secuencia.

Si se ha decidido tratar a los separadores como entidades de longitud variable en la estructura del índice, puede ahorrarse espacio colocando el separador más corto en la estructura del índice.

Cuando se usan los separadores como un mapa para el conjunto de secuencias, debe decidirse extraer el bloque que está a la derecha del separador o bien el de la izquierda, de acuerdo a la siguiente regla:

Llave < separador	Ir a la izquierda
Llave == separador	Ir a la derecha
Llave > separador	Ir a la derecha

Arboles $B^{\pm}$ de prefijos simples

Los separadores pueden colocarse en forma de índice de un árbol B para los bloques del conjunto de secuencias. Al índice en forma de árbol B se le llama **conjunto índice**, y este y el conjunto de secuencias forman una estructura de archivos llamada **árbol B^+ de prefijos simples**. El modificador prefijos simples indica que el conjunto índice contiene los separadores más cortos, o prefijos de las llaves, en lugar de copias de las llaves verdaderas.

Los árboles B^+ de prefijos simples tienen las siguientes propiedades:

- Cada página tiene como máximo M descendientes.
- Cada página, menos la raíz y las hojas, tienen entre [M/2] y M hijos.
- La raíz tiene al menos dos descendientes (o ninguno)
- Todas las hojas aparecen en igual nivel.
- Una página que no sea hoja si tiene K descendientes contiene K-1 llaves.
- Los nodos terminales representan un conjunto de datos y son linkeados juntos.

Los nodos terminales no contienen datos sino punteros a los datos.

Mantenimiento de árboles $B^{\pm}$ de prefijos simples

Las inserciones y eliminaciones en el conjunto índice se manejan como operaciones estándar de árboles B, el hecho de que ocurra división o inserción sencilla, concatenación o eliminación simple, depende por completo de que tan lleno esté el nodo del conjunto índice.

La inserción y eliminación de registros siempre tiene lugar en el conjunto de secuencias, ya que es allí donde están los registros. Si la división, concatenación o redistribución son necesarias, la operación se efectúa como si el conjunto de secuencias no existiera. Entonces, después de que terminan las operaciones de los registros en el conjunto de secuencias, se hacen los cambios necesarios en el conjunto índice:

- Si los bloques se dividen en el conjunto de secuencias, debe insertarse un nuevo separador dentro del conjunto índice.
- Si se concatenan bloques en el conjunto de secuencias, debe eliminarse un separador del conjunto índice.
- Si los registros se redistribuyen entre bloques en el conjunto de secuencias, debe cambiarse el valor de un separador en el conjunto índice.

Tamaño de bloque del conjunto índice

El tamaño físico de un nodo para el conjunto índice es, por lo regular, el mismo que el tamaño físico de un bloque en el conjunto de secuencias. Existen varias razones para usar un tamaño común de bloque para los conjuntos índice y de secuencias:

- Un tamaño de bloque común facilita la implantación de un esquema de manejo de buffers para crear un árbol B^+ de prefijos simples virtual, parecido a los árboles B virtuales.
- Los bloques del conjunto índice y los del conjunto de secuencias frecuentemente se incorporan dentro del mismo archivo para evitar desplazamientos en dos archivos separados cuando se tiene acceso a un árbol B^+ de prefijos simples. El uso de un archivo para ambos tipos de bloques es más sencillo si los tamaños de bloques son iguales.

Estructura interna de los bloques del conjunto índice: árbol B de orden variable

Un **árbol B** es **de orden variable** cuando el número de descendientes directos de cualquier nodo del árbol es variable. Esto ocurre cuando los nodos del árbol B contienen un número variable de llaves o descendientes. Esta forma es la que se usa con mayor frecuencia cuando existe variabilidad en las longitudes de las llaves o los separadores. Los árboles B^+ de prefijos simples siempre hacen uso de un árbol B de orden variable como conjunto índice, de modo que es posible aprovechar la compresión de los separadores y colocar más de ellos en un bloque.

Hay muchas formas de combinar la lista de separadores, los índices de los separadores y la lista de NRB (número relativo de bloque) dentro de un solo bloque del conjunto índice. Un método posible es: además del vector de separadores, índice y la lista de números de bloque asociados, esta estructura de bloques incluye un contador de separadores y la longitud total de los separadores. El contador de separadores se necesita para ayudar a encontrar el elemento del medio en el índice de los separadores, de tal forma que pueda empezarse la búsqueda binaria. Como el índice de los separadores comienza al final de la lista de longitud variable, se necesita conocer su tamaño para encontrar el inicio del índice.

Carga de un árbol B^+ de prefijos simples

Una forma de construir un árbol B^+ de prefijos simples es a través de una serie de inserciones sucesivas. La dificultad con este método es que la división y la redistribución son relativamente costosas. En vez de eso puede comenzarse por clasificar los registros que se van a cargar, para así garantizar que el siguiente registro que se encuentre sea el siguiente registro que se requiera cargar.

Al trabajar a partir de un archivo clasificado, pueden colocarse los registros en los bloques del conjunto de secuencias, uno por uno, comenzando un bloque nuevo cuando se llena el que se está trabajando. A medida que se hace la transición entre dos bloques del conjunto de secuencias se puede ir determinando el separador más corto para los bloques. Estos separadores pueden agruparse dentro de un bloque del conjunto índice, que se construye y almacena en memoria RAM hasta que se llene.

La principal ventaja de clasificar antes el archivo es que el proceso de carga es más rápido, ya que:

- La salida puede escribirse en forma secuencial.
- Se realiza sólo un paso sobre los datos, en lugar de los numerosos pasos asociados con las inserciones aleatorias.
- No es necesario reorganizar ningún bloque a medida que se avanza.

Hay dos ventajas más sobre el uso de un proceso de carga separado:

- La carga secuencial proporciona más control sobre la cantidad y la colocación del espacio vacío en el árbol recién cargado.
- Si se usa el mismo archivo para los bloques del conjunto de secuencias y los del conjunto índice, se garantiza que un bloque del conjunto índice se inicie en proximidad física con los bloques del conjunto de secuencias que son sus descendientes. Esto puede minimizar los desplazamientos cuando se realiza una búsqueda hacia abajo en el árbol.

Arboles B^+

La diferencia entre un árbol B^+ de prefijos simples y un árbol B^+ sencillo es que esta última estructura no implica el uso de prefijos como separadores, sino que los separadores del conjunto índice son simplemente copias de las llaves reales.

Las operaciones efectuadas sobre árboles B^+ son las mismas que las de los árboles B^+ de prefijos simples. Tanto los árboles B^+ como los árboles B^+ de prefijos simples consisten en un conjunto de secuencias, y se acoplan con un conjunto índice que proporciona acceso rápido al bloque que contiene alguna combinación llave/registro en particular. La única diferencia es que en el árbol B^+ de prefijos simples se construye un conjunto índice con los separadores más cortos, formados a partir de los prefijos de las llaves.

Factores para usar B^+ y no B^+ de prefijos simples:

- Por más que los separadores de B^+ con prefijos simples son más cortos, tenemos que tener una estructura que es más costosa mantener y más compleja ya que tenemos que tener un conjunto índice que maneje campos de longitud variable.
- Con algunos conjuntos de llaves no se logra la compresión necesaria para usar prefijos simples para producir separadores.

Comparación entre los árboles B, $B^{\pm}$ y $B^{\pm}$ de prefijos simples

Estas tres herramientas comparten las siguientes características:

- Son estructuras paginadas de índice, lo que significa que llevan bloques completos de información a la memoria RAM a la vez.
- Mantienen arboles de altura. Es decir, los árboles no crecen en forma dispareja.
- Los árboles crecen de abajo hacia arriba. El balance se mantiene por medio de división de bloques, concatenación y redistribución.
- Con las tres estructuras es posible obtener mayor eficiencia de almacenamiento usando división de dos a tres y redistribución en lugar de la división de bloques, cuando sea posible.
- Los tres métodos pueden implementarse como estructuras de árboles virtuales, donde los bloques cuyo uso es más reciente se almacenan en memoria RAM.
- Cualquiera de estos métodos se puede adaptar para usarse con registros de longitud variable.

Existen dos ventajas significativas de la estructura de árboles B^+ sobre la de árboles B:

- El conjunto de secuencias puede procesarse en una forma verdaderamente lineal y secuencial, proporcionando acceso eficiente a los registros en el orden de la llave.
- El uso de separadores en lugar de registros completos en el conjunto índice con frecuencia significa que el número de separadores que pueden colocarse en un solo bloque del conjunto índice en un árbol B^+ excede en forma sustancial al número de registros que podrían colocarse en un bloque del mismo tamaño en un árbol B. Dado que pueden colocarse más separadores en un bloque de tamaño dado, se deduce que el número de otros bloques descendientes de ese puede ser mayor. Como consecuencia, el método de árbol B^+ con frecuencia puede producir un árbol de menor altura que el que daría el método de árbol B.

El árbol B^+ de prefijos simples aprovecha la segunda ventaja haciendo los separadores del conjunto índice más pequeños que las llaves del conjunto de secuencias, en lugar de usar solo copias de estas llaves. Si los separadores son menores, entonces puede haber más en un bloque y obtener así un número aún más alto de ramas.

Para obtener esta compresión de separadores y el incremento en el número de ramas debe usarse una estructura de conjunto índice que maneje campos de longitud variable.

Unidad 14

Hashing (dispersión)

Una función de dispersión h(K), transforma una llave K en una dirección. La dirección resultante se usa como base para la búsqueda y el almacenamiento de registros.

La dispersión se asemeja a la indización en cuanto a que implica la asociación de una llave con una dirección relativa de registro, pero difiere de ella en dos aspectos:

- Con la dispersión, las direcciones generadas parecen ser aleatorias; no hay conexión obvia inmediata entre la llave y la localidad del registro correspondiente, aun cuando la llave se usa para determinar la colocación del registro. Por tal motivo, a la dispersión se la denomina algunas veces **aleatorización**.
- Con la dispersión, dos llaves diferentes pueden transformarse en la misma dirección, de modo que los dos registros pueden enviarse al mismo lugar del archivo. Cuando esto ocurre, se le llama **colisión**, y debe encontrarse un medio de resolverla. A las llaves que por dispersión se convierten en la misma dirección se las denomina **sinónimos**.

Algoritmo simple de dispersión

Uno de los objetivos en la elección de cualquier algoritmo de dispersión debe ser esparcir los registros tan uniformemente como sea posible en el intervalo de direcciones disponible. También se debe lograr que las llaves sean independientes, no influyan una sobre la otra.

Este algoritmo tiene 3 pasos:

1. Representar la llave en forma numérica.
2. Desglosar y sumar.
3. Dividir entre un número primo y usar el residuo como dirección.

Paso 1: representar la llave en forma numérica. Si la llave ya es un número, entonces este paso ya está hecho. Si es una cadena de caracteres, se usa el código ASCII de cada carácter para formar un número.

Paso 2: desglosar y sumar significa tomar pedazos del número y sumarlos. Antes de sumar los números, debe mencionarse el problema de que en la mayoría de los casos los tamaños de los números que pueden sumarse están limitados.

Se debe identificar primero el mayor valor individual que se vaya a agregar en la suma, y asegurándose después de cada paso que el resultado intermedio difiera del valor máximo por esa cantidad.

Puede asegurarse en este algoritmo que ningún resultado intermedio exceda de lo permitido utilizando el operador mod.

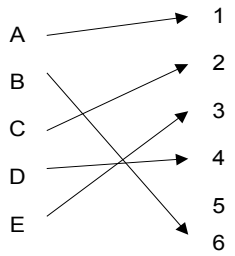
Paso 3: dividir entre el tamaño del espacio de direcciones. El propósito de este paso es recortar el número producido en el paso 2, para que esté dentro del intervalo de direcciones de registros en el archivo. Se puede hacer esto dividiendo ese número entre el tamaño de las direcciones del archivo; el residuo será la dirección base del registro.

Si s representa la suma producida en el paso 2, n representa el divisor (número de direcciones en el archivo), y a representa la dirección que se está intentando producir, se aplica la fórmula: a = s mod n. El residuo producido por el operador será un número entre 0 y n - 1.

Es común usar un número primo como divisor porque los números primos tienden a distribuir residuos en forma mucho más uniforme que los que no lo son.

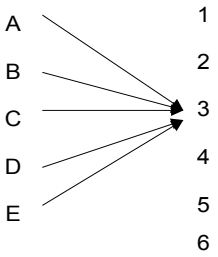
```
función dispersión (llave, maxdir)
        sum := 0
        j := 0
        mientras (j < # elementos de llave)
                sum := sum + 100 * llave[j] + llave[j + 1]
                j := j + 2
        fin mientras
        retorna (sum mod maxdir)
```

Funciones de dispersión



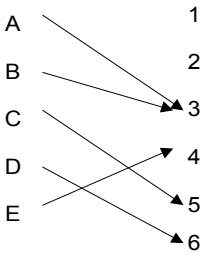
uniforme

sin sinónimos



peor

todos sinónimos



aceptable

pocos sinónimos

Aunque una distribución aleatoria de registros entre las direcciones disponibles no es lo ideal, es una alternativa aceptable, dado que es prácticamente imposible encontrar una función que logre una distribución uniforme. Las distribuciones uniformes pueden descartarse totalmente, pero a veces pueden encontrarse mejores distribuciones que las aleatorias, en el sentido de que mientras generen un adecuado número de sinónimos, distribuyen los registros entre las direcciones más uniformemente que una distribución aleatoria.

Estos son algunos métodos que son potencialmente mejores que el aleatorio:

- Examinar las llaves para buscar un patrón (Análisis de dígitos): algunas veces las llaves exhiben patrones que se esparcen en forma natural. Esto sucede con mayor probabilidad entre llaves numéricas que entre llaves alfabéticas. Si alguna parte de la llave muestra un patrón útil, puede usarse una función de dispersión que extraiga esa parte de la llave.
- Desglosar partes de la llave (Plegado y suma): los dígitos extremos de la llave se pliegan como una hoja de papel. Luego se suman y se ajustan al espacio de direcciones. Este método destruye los patrones originales de las llaves, pero en determinadas circunstancias puede preservar la separación entre ciertos subconjuntos de llaves que se esparcen por sí mismas en forma natural. Ej: # máximo de dir =100. Llave: 621 123 130 → 123 + 126 + 031 = 280 mod 100 = 80. Dirección base: 80.
- División: la llave se divide por un número aproximadamente igual al número de direcciones disponibles. Como dirección se toma el resto de la operación. Es más probable que la división entre un número primo genere resultados diferentes a partir de diferentes secuencias consecutivas que los obtenidos con la división entre un número que no sea primo.
  Ej: llave = 1224. 100 ≃ 101 y 101 es primo → 1224 mod 101 = 12. Dirección base: 12.

Los métodos anteriores se idearon para sacar partido del orden natural de las llaves. Los dos métodos siguientes deben intentarse cuando, por alguna razón, los métodos mejores que el aleatorio no funcionan. En estos casos, la aleatoriedad es el objetivo.

- Centros cuadrados: se eleva la llave al cuadrado y se extrae del centro del resultado el número de dígitos que sea necesario. Mientras las llaves no contengan muchos ceros adelante o atrás, este método por lo regular produce resultados razonablemente aleatorios. $1224^2 = 1498176$ → 981 mod 100 = 81
- Transformación de la base (Conversión de la raíz): este método implica convertir la llave a alguna otra base numérica que no sea con la que se esté trabajando, y tomar después el resultado del módulo con la máxima dirección como la dirección de dispersión. Ej: 1224_8 = 660 mod 100 = 60. Dirección base: 60.

La transformación de base por lo general es más confiable que el método del centro cuadrado para aproximarse a lo verdaderamente aleatorio, aunque se ha encontrado que el centro cuadrados da buenos resultados cuando se aplica a algunos conjuntos de llaves.

- División polinómica: cada dígito de la llave se toma como un coeficiente de un polinomio que luego se divide por otro polinomio fijo. Los coeficientes del resto se toman como dirección base. Ej:
  172148 → $17x^2 + 21x + 48$ | P(x) R = ax + b
  R(x) C(x) Dirección base = ab.

Densidad de empaquetamiento

El término densidad de empaquetamiento se refiere a la proporción entre el número de registros por almacenar (r) y el número de espacios disponibles (N).

$$\frac{\text{Número de registros}}{\text{Número de espacios}} = \frac{r}{N} = \text{Densidad de empaquetamiento}$$

La densidad de empaquetamiento es una medida de la cantidad del espacio que se usa en realidad en un archivo, y es el único valor necesario para evaluar el desempeño en un ambiente de dispersión, suponiendo que el método de dispersión usado logra una distribución de registros razonablemente aleatoria. No importan el tamaño real del archivo ni su espacio de direcciones; lo importante es el tamaño relativo de los dos, que están dados por la densidad de empaquetamiento.

Es necesario decidir cuánto espacio se esta dispuesto a desperdiciar para reducir el número de colisiones.

Estimación de colisiones para diferentes densidades de empaquetamiento

Es necesario poder predecir el número de colisiones probables para una densidad de empaquetamiento determinada. La función de Poisson proporciona la herramienta para lograrlo.

$$p(x) = \frac{(r/N)^x e^{-r/N}}{x!}$$

donde r es el número de registros a almacenar, N es el número de espacios disponibles y x es el número de registros asignados a una dirección dada. Entonces p(x) indica la probabilidad de que a una dirección determinada se hayan asignado x registros luego de haber aplicado la función de dispersión a los N registros.

Supongamos que se asignan 1000 direcciones para guardar 500 registros en un archivo dispersado en forma aleatoria, y cada dirección puede almacenar un registro. La densidad de empaquetamiento del archivo es 500/1000 = 0.5.

¿Cuántas direcciones no deberán tener registros asignados? Como p(0) da la proporción de direcciones sin registros asignados, el número de tales direcciones es Np(0) = 607.

¿Cuántas direcciones deben tener exactamente un registro asignado (no hay sinónimos)? Np(1)=303.

¿Cuántas direcciones deben tener un registro más uno o varios sinónimos? Los valores de p(2), p(3), p(4), y demás, dan las proporciones de direcciones con uno, dos, tres y más sinónimos asignados a ella. Por tanto, la suma p(2) + p(3) + p(4) + ... da la proporción de todas las direcciones que tienen un sinónimo al menos.

El número de direcciones con uno o más sinónimos es sólo el producto de N y resulta en: N[p(2)+p(3)+p(4)+p(5)] = 90. Se toma solamente hasta x igual a 5 ya que de aquí en adelante son números muy pequeños.

Suponiendo que sólo puede asignarse un registro a cada dirección base, ¿cuántos registros en saturación pueden esperarse? Para cada una de las direcciones representadas por p(2) puede almacenarse un registro en la dirección, y el otro debe ser un registro en saturación. Para cada dirección representada por p(3), un registro puede almacenarse en la dirección, y dos son registros en saturación, y así sucesivamente. Por lo tanto el número de registros en saturación está dado por

1 x N x p(2) + 2 x N x p(3) + 3 x N x p(4) + 4 x N x p(5) = 107.

¿Qué porcentaje de registros deben ser registros en saturación? Si hay 107 registros en saturación y 500 registros en total, entonces la proporción de registros en saturación es 107/500 = 0.214 = 21.4%.

Si la densidad de empaquetamiento es del 50% y cada dirección puede almacenar sólo un registro, puede esperarse que alrededor del 21% de todos los registros sean almacenados en algún otro lugar que no sean sus direcciones base.

Cuanto mayor es el número de registros sin lugar propio, mayor es la competencia por el espacio con los otros registros sin lugar. Pronto pueden formarse cúmulos de registros en saturación, que en algunos casos conducen a búsquedas extremadamente lentas para algunos de los registros.

Tratamiento de colisiones

Aun cuando un algoritmo de dispersión sea muy bueno, es probable que ocurran colisiones. Por lo tanto, cualquier programa de dispersión debe incorporar algún método para tratar con los registros que no pueden entrar en su dirección base.

Saturación progresiva

Cuando se produce una colisión se busca en las siguientes direcciones en secuencia hasta que se encuentra una vacía. La primera dirección vacía se convierte en la dirección del registro.

Cuando se quiere buscar un registro se comienza por la dirección producida y se continúa de manera secuencial hasta encontrar el registro buscado.

Cuando se busca un espacio libre o un registro al final del archivo, se da la vuelta al espacio de direcciones del archivo y se elige la dirección 0 como siguiente dirección.

Si se busca un registro que nunca se colocó en el archivo, la búsqueda comienza en la dirección base del registro, luego se continúa buscando en las localidades sucesivas. Pueden suceder dos cosas:

- Si se encuentra una dirección vacía, la rutina de búsqueda puede suponer que esto significa que el registro no está en el archivo.
- Si el archivo está lleno, la búsqueda vuelve a donde comenzó. Solo entonces está claro que el registro no está en el archivo. Cuando esto ocurre, o incluso cuando se está llenando el archivo, la búsqueda no puede llegar a ser intolerablemente lenta, esté o no el registro que se busca dentro del archivo.

La gran ventaja de la saturación progresiva es su simplicidad.

Longitud de búsqueda

La razón de evitar la saturación es que ocurren búsquedas adicionales cuando un registro no se encuentra en su dirección base. Si hay muchas colisiones, habrá bastantes registros en saturación que ocupen espacios que no deben. Pueden formarse cúmulos de registros, lo que da por resultado la colocación de registros a gran distancia de su dirección base, de tal forma que se requieren muchos accesos al disco para extraerlos.

El término longitud de búsqueda se refiere al número de accesos requerido para extraer un registro de la memoria secundaria. En el contexto de la dispersión, la longitud de búsqueda se incrementa cada vez que ocurre una colisión. Si un registro se encuentra a gran distancia de su dirección base, la longitud de búsqueda puede ser inaceptable. Un buen indicador del problema de la saturación es la longitud media de búsqueda, que no es sino el número promedio de veces que se espera acceder al disco para extrae un registro. Una primera estimación de la longitud media de búsqueda puede calcularse mediante la longitud total de búsqueda (la suma de las longitudes de búsqueda de los registros individuales) dividida entre el número de registros:

$$\text{Longitud media de búsqueda} = \frac{\text{Longitud total de búsqueda}}{\text{Número total de registros}}$$

Usando la saturación progresiva, la longitud media de búsqueda aumenta rápidamente conforme se incrementa la densidad de empaquetamiento.

Las longitudes medias de búsqueda mayores de 2.0 por lo general se consideran inaceptables.

Almacenamiento de más de un registro por dirección: compartimientos

Se usa la palabra compartimiento para describir un bloque de registros que se extraen en un acceso al disco, en especial cuando se consideran compartiendo la misma dirección. En discos que se hace referencia por sector, un compartimiento por lo regular consiste en uno o más sectores; en discos que hacen referencia por bloques, un compartimiento puede ser un bloque.

Cuando un registro se almacena o se extrae, su dirección de compartimiento base se determina por dispersión. El compartimiento completo se carga en memoria primaria. Entonces puede usarse una búsqueda en memoria RAM a través de registros sucesivos en el compartimiento para encontrar el registro deseado. Cuando un compartimiento se llena, persiste el problema de la saturación, pero esto ocurre con mucho menor frecuencia cuando se usan compartimientos que cuando cada dirección puede almacenar sólo un registro.

Cuando se usan compartimientos, la fórmula usada para calcular la densidad de empaquetamiento cambia porque cada dirección de compartimiento puede almacenar más de un registro. Para calcular cuánto

empaquetamiento hay en un archivo, es necesario considerar tanto el número de direcciones (compartimientos) como el número de registros que pueden colocarse en cada dirección (tamaño del compartimiento). Si N es el número de direcciones y b es el número de registros que caben en el compartimiento, entonces bN es el número de localidades disponibles para registros. Si r es el número de registros del archivo, entonces

Densidad de empaquetamiento = r/bN

La densidad de empaquetamiento no cambia. La clave de la mejora es que, aunque hay menos direcciones, cada dirección individual tiene más espacio para variación del número de registros que se le asignan.

Cuando hay compartimientos hay más sinónimos que cuando no los hay, pero la mitad de esos sinónimos no provoca registros en saturación porque cada compartimiento puede almacenar dos registros.

A medida que el tamaño del compartimiento aumenta, el desempeño mejora. ¿Qué tan grandes deben ser los compartimientos? Depende en buena parte de varias características del sistema, entre ellas los tamaños de los buffers que el sistema operativo maneje, las capacidades de sectores y pistas de los discos, y los tiempos de acceso del equipo.

Como regla general, probablemente no sea buena idea usar compartimientos mayores de una pista (a menos que los registros sean muy grandes). Sin embargo, aun una pista puede ser algunas veces demasiado grande cuando se considera el tiempo que toma transmitir unos cuantos sectores. Como la dispersión casi siempre implica la extracción de un solo registro por búsqueda, cualquier tiempo de transmisión adicional que resulte del uso de compartimientos extragrandes es esencialmente un gasto inútil.

En muchos casos un cúmulo es el mejor tamaño para un compartimiento.

La longitud media de búsqueda representa el número promedio de compartimientos a los que debe accederse para extraer un registro. Cuanto más grande es el compartimiento, menor es la longitud de búsqueda.

Eliminación

Cuando se efectúa una eliminación no debe permitirse que el espacio liberado por la eliminación obstaculice las búsquedas posteriores y debe ser posible reutilizar el espacio liberado para las adiciones posteriores.

Cuando se usa la saturación progresiva, la búsqueda de un registro termina si se encuentra una dirección vacía. Por ello, no es conveniente dejar direcciones vacías que terminen la búsqueda por saturación en forma inapropiada.

Marcas de inutilización

Una técnica sencilla usada para identificar registros eliminados reemplaza el registro eliminado (o solo su llave) con un marcador que indique que alguna vez hubo un registro, pero ya no. Tal marcador se denomina marca de inutilización. El uso de marcas de inutilización resuelve los dos problemas anteriores.

No es necesario insertar marcas de inutilización cada vez que ocurre una eliminación.

Con el uso de las marcas de inutilización, la inserción de registros se vuelve más difícil puesto que los programas que efectúan el cargado inicial únicamente buscan la primera ocurrencia de un espacio vacío, ahora se puede insertar un registro en donde ocurran como llave tanto el signo de vacío como el de eliminado.

Esta nueva característica, provoca una mayor longitud media de búsqueda.

El uso de marcas de inutilización permite que los algoritmos de búsqueda trabajen y ayuda en la recuperación de almacenamiento, pero se puede esperar cierto deterioro en el desempeño después de varias eliminaciones e inserciones en el archivo.

En general, después de un gran número de inserciones y eliminaciones puede esperarse encontrar muchas marcas de inutilización ocupando lugares que podrían ser ocupados por registros cuyas direcciones base los preceden, pero que están almacenados después de ellas. Cada marca de inutilización representa una oportunidad desaprovechada de reducir en uno el número de localidades que deben examinarse mientras se buscan estos registros.

Existen tres tipos de soluciones para el problema del deterioro de las longitudes medias de búsqueda. Una implica reorganizar de forma local cada vez que suceda una eliminación. Otra solución implica la reorganización completa del archivo después de que la longitud media de búsqueda alcanza un valor inaceptable. Una tercera posibilidad es usar un algoritmo de resolución de colisiones por completo diferente.

Otras técnicas de tratamiento de colisiones

Dispersión doble

Un método para evitar el acumulamiento consiste en almacenar los registros en saturación a una gran distancia de sus direcciones base mediante dispersión doble, lo cual significa que, cuando sucede una colisión, se aplica una segunda función de dispersión a la llave para producir un número c que es el primo relativo (c y N son primos relativos si no tienen divisores comunes) al número de direcciones. El valor c se agrega a la dirección base para producir la dirección en saturación. Si ya está ocupada, se agrega c para producir otra dirección en saturación. Este procedimiento continúa hasta que se encuentra una dirección que este libre.

La dispersión doble tiende a esparcir los registros en el archivo pero viola la localidad al trasladar deliberadamente los registros en saturación a cierta distancia de sus direcciones base, incrementando así la probabilidad de que el disco necesite tiempo adicional para obtener la nueva dirección en saturación. Los programas con dispersión doble pueden resolver este problema si son capaces de generar direcciones en saturación de tal modo que los registros en saturación se mantengan en el mismo cilindro que los registros con direcciones base.

Saturación progresiva encadenada

Funciona de la misma manera que la saturación progresiva, excepto que los sinónimos se ligan con apuntadores. Esto es, cada dirección base contiene un número que indica el lugar del siguiente registro con la misma dirección base. El siguiente registro contiene, a su vez, un apuntador al siguiente registro con la misma dirección base, y así sucesivamente. El efecto de esto es que para cada conjunto de sinónimos hay una lista ligada que conecta sus registros, y es esta lista en la que se busca cuando se requiere un registro.

La ventaja de la saturación progresiva encadenada sobre la sencilla es que en cualquier búsqueda solo se necesita acceder a los registros con llaves que son sinónimos.

El uso de la saturación progresiva encadenada implica que se debe agregar un campo de liga a cada registro, lo que requiere un poco más de almacenamiento. Además, un algoritmo de encadenamiento debe garantizar la posibilidad de llegar a cualquier sinónimo partiendo de su dirección base.

Un problema es que una dirección que debe ser ocupada por un registro base está ocupada por un registro diferente. Una solución es procurar que toda dirección que se califique como dirección base para algún registro del archivo realmente almacene un registro base. El problema es fácil de manejar si el archivo se carga desde el principio usando una técnica de carga en dos pasos.

La carga en dos pasos, implica cargar el archivo en dos pasos. En el primer paso sólo se cargan los registros con direcciones base. Los registros que no son base se guardan en un archivo separado. Esto garantiza que ninguna dirección base potencial este ocupada por registros en saturación. En el segundo paso se carga y almacena cada registro en saturación en una de las direcciones libres, de acuerdo con la técnica de solución de colisiones que se esté usando.

La carga en dos pasos garantiza que toda dirección base potencial sea en realidad una dirección base, de modo que se resuelve el problema. Sin embargo, no se garantiza que las eliminaciones e inserciones posteriores no vuelvan a crear el mismo problema. Mientras se usa el archivo para almacenar registros en direcciones base y también en saturación, permanecerá el problema de que los registros en saturación desplacen a los nuevos que se asignan a una dirección ocupada por uno en saturación.

Encadenamiento con un área de saturación separada

Una forma de evitar que los registros en saturación ocupen las direcciones base en donde no deben estar es moverlos a un área de saturación separada. Al conjunto de direcciones base se le llama área principal de datos, y al conjunto de direcciones de saturación se le llama área de saturación. La ventaja de este enfoque es que mantiene todas las direcciones base potenciales que no se han usado libres para inserciones posteriores.

Cuando se agrega un nuevo registro no ocurre ningún problema. Si hay lugar para su dirección base, se almacena ahí. Sino, se mueve al archivo de saturación, donde se agrega a la lista ligada que comienza en la dirección base.

El uso de un área de saturación separada simplifica un poco el proceso, y parecería que mejora el desempeño, en especial cuando ocurren muchas inserciones y eliminaciones. Sin embargo, no siempre es así. Si el área de saturación separada está en un cilindro diferente del de la dirección base, toda búsqueda de un registro en saturación implicará un movimiento de cabeza muy costoso.

Una situación en la que se requiere un área de saturación separada se presenta cuando la densidad de empaquetamiento es mayor a uno: hay mas registros que direcciones base.

Variantes de encadenamiento

Hay tres técnicas para la solución de saturación encadenada:

- Listas separadas: pone la lista de registros en saturación en la cubeta. Solo los registros de la lista son examinados en una búsqueda, se minimizan las comparaciones.

Si se borra un registro base de la cubeta, se reemplaza por uno de la lista. Requiere mucho espacio para los punteros.

- Listas unidas: almacenar los registros en espacio separado. Cada cubeta contiene un puntero apuntando a la próxima cubeta. Los punteros establecen los registros en saturación.
- Directorios: el espacio de las cubetas es mas del que se necesita, entonces el espacio restante se utiliza para almacenar punteros a los registros en saturación de la cubeta y sus llaves.

Tablas de dispersión

Supóngase que se tiene un archivo de dispersión que no contiene registros, sólo apuntadores a registros. El archivo sólo es un índice donde se busca por dispersión en lugar de usar otro método. Este método de organización de archivos se conoce como tabla de dispersión.

La organización por tablas de dispersión ofrece muchas de las ventajas que por lo regular proporciona la indización simple, con la ventaja adicional de que la búsqueda del índice mismo requiere sólo un acceso. El archivo de datos puede ser de diferentes formas. Por ejemplo, puede consistir en un conjunto de listas ligadas de sinónimos, un archivo clasificado, o un archivo de entradas secuenciales. Además, las tablas de dispersión manejan convenientemente el uso de registros de longitud variable.

Hashing estático

Se debe predeterminar el número de direcciones (N). Como N es fijo, si el archivo sobrepasa a N, se debe redispersar todo el archivo con un nuevo espacio de direcciones.

Si N es grande y la cantidad de registros a almacenar es pequeña, entonces se desperdicia mucho espacio.

Si N es pequeño, habrá muchas colisiones, baja el rendimiento.

Podría usarse para archivos que no tengan muchas inserciones posteriores.

Los registros son de longitud fija.

Hashing dinámico

No es necesario predeterminar el tamaño de direcciones, en lugar de esto, se tiene una tabla que crece o decrece conforme crece o decrece el archivo. La tabla tiene apuntadores a registros y además una cabecera que indica el tamaño actual de la tabla. Si un registro no entra en una dirección, se debe multiplicar el tamaño de la cubeta. Si la tabla se redispersa de n a 2n, los registros que originalmente se dispersaban a la misma dirección, ahora lo hacen a i o a i+n y ningún registro de otros compartimientos se dispersa a i o a i+n. Se deben dividir solo aquellos compartimientos que contengan registros en saturación, por lo que se desperdicia muy poco tiempo al reorganizar la tabla original.

El rendimiento se mantiene conforme crece el archivo. Se puede usar cuando no sabemos cuanto va a crecer el archivo.

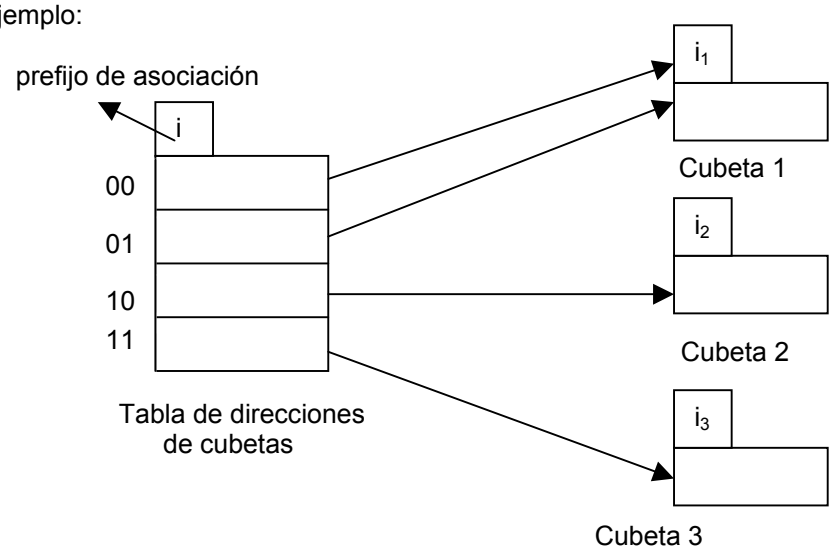
Hashing extensible

La asociación extensible maneja los cambios en el tamaño de la base de datos dividiendo y fusionando cubetas conforme la base de datos crece y se reduce. Como resultado se mantiene la eficiencia de espacio. Además, puesto que la reorganización se realiza cada vez únicamente en una cubeta, el tiempo extra requerido es aceptablemente bajo.

Con la asociación extensible elegimos una función de asociación h con las propiedades deseables de uniformidad y aleatoriedad. Sin embargo, esta función de asociación genera valores en un rango relativamente grande de enteros binarios de b bits. Un valor típico de b es 32.

Inicialmente no usamos todos los b bits de la asociación. En un momento dado, usamos i bits, donde $0 \leq i \leq b$. Estos i bits representan una posición relativa en una tabla adicional de direcciones de cubetas. El valor de i aumenta y disminuye con el tamaño de la base de datos.

Ejemplo:



Prefijo de asociación: se requieren i bits de la asociación h(K) para determinar la cubeta correcta para K. El entero asociado con la cubeta j se presenta como i_j.

El número de entradas de la tabla de direcciones de cubetas que apuntan a la cubeta j es $2^{(i - i_j)}$.

Para **localizar** la cubeta que contiene el valor de la clave de búsqueda K_l, tomamos los primeros i bits más significativos de $h(K_l)$, vemos la entrada de la tabla que corresponde a esta cadena de bits, y seguimos el puntero de la cubeta en la entrada de la tabla.

Para **insertar** un registro, seguimos el mismo procedimiento hasta llegar a la cubeta (j). Si hay lugar en la cubeta, insertamos la información y después insertamos el registro en el archivo. Si la cubeta está llena, debemos partirla y redistribuir los registros actuales más el nuevo. Para partir la cubeta, primero debemos determinar si necesitamos aumentar el tamaño de bits que usamos en la asociación.

- Si $i = i_j$, entonces solamente una entrada en la tabla de direcciones de cubetas apunta a la cubeta j. Por lo tanto, necesitamos aumentar el tamaño de la tabla de direcciones de forma que podamos incluir punteros en las dos cubetas que resultan de la partición de la cubeta j. Incrementamos el valor de i en 1, duplicando así el tamaño de la tabla. Ahora dos entradas en la tabla de direcciones apuntan a la cubeta j. Asignamos una nueva cubeta (z) y hacemos que la segunda entrada apunte a la nueva cubeta. Ponemos i_j e i_z a i. Ahora se asocia cada registro de la cubeta j y, dependiendo de los primeros i bits (habíamos añadido 1), se guarda en la cubeta j o en la z. Se puede llegar a partir otra cubeta si todos los registros tienen el mismo prefijo de asociación.
- Si $i > i_j$, entonces más de una entrada en la tabla de direcciones apunta a la cubeta j. Así, podemos partir la cubeta j sin aumentar el tamaño de la tabla. Ahora, todas las entradas que apuntan a la cubeta j corresponden a prefijos de asociación que tienen el mismo valor en los bits i_j más a la izquierda. Asignamos una nueva cubeta (z) y ponemos i_j e i_z al valor que resulta de añadir 1 al valor original de i_j. A continuación necesitamos ajustar las entradas en la tabla de direcciones que anteriormente apuntaban a la cubeta j. Dejamos la primera mitad de las entradas como estaban (apuntando a j) y ponemos todas las demás entradas apuntando a la cubeta z. Luego se reasocia cada registro de la cubeta j y se reasigna a la cubeta j o la z. Se reintenta la inserción y si vuelve a fallar (poco probable) se aplica uno de los dos casos, $i = i_j$ o $i > i_j$.

Para **eliminar** un registro, se realiza el mismo procedimiento de búsqueda hasta llegar a la cubeta (j). Sacamos la clave de búsqueda de la cubeta y el registro del archivo. La cubeta también se elimina si queda vacía. En este caso se pueden unir varias cubetas y que el tamaño de la tabla de direcciones se puede partir en dos.

Diferencias y similitudes entre hashing estático y dinámico

En ambos la función de dispersión debe distribuir los registros en forma uniforme y aleatoria. Ambos utilizan cubetas para disminuir el número de colisiones.

Diferencias:

- En el hash estático el número de cubetas es fijo (se conoce de antemano). En el dinámico varía según crezca la base de datos, se aprovecha mejor el espacio.
- En el hash estático las inserciones, eliminaciones y búsquedas son más rápidas. En el dinámico las búsquedas requieren un nivel de indirección especial, ya que debemos tener un acceso a la tabla de direcciones antes de acceder a la cubeta en sí.
- El hash estático lo utilizaría cuando conozco de antemano la cantidad máxima de registros que voy a almacenar, que no va a crecer más que eso. El hash dinámico, lo utilizo cuando tengo una base de datos cuyo tamaño varía entre límites que no conozco.